503
COMPUTER
MANUAL

VOL. 2
C

503 COMPUTER MANUAL

503 TECHNICAL MANUAL

VOLUME 2: PROGRAMMING INFORMATION

CONTENTS LIST

PART 3: THE INTEGRATED SOFTWARE SYSTEM FOR THE NON-BASIC 503

SECTION 1: INTRODUCTION

CONTENTS

1. INTRODUCTION

This introduction provides an overall picture of the software system provided for the 503 Computer. The system comprises several interdependent programs, each concerned with a particular aspect of computing techniques. Taken together, the programs form a system designed to simplify the programmer's task while at the same time facilitating maximum utilisation of computer time and efficiency. Reference should be made to the technical specifications (see Parts 4 and 5) for details of implementation and use.

2. THE SYSTEMS PROGRAMS

The programming and operating aids may be classified broadly under five heads; they are, however, fully integrated and interdependent, and are discussed individually in the following sections under these headings:

The Operating System

The Time-sharing System

The Storage Control System

The Program Checking System

The Programming Languages

3. CONFIGURATION

The 503 software system is designed for use on a configuration which includes some form of auxiliary storage. On a basic computer (i.e. a configuration which may include peripheral devices but has no form of auxiliary storage) the Reserved Area Program, ALGOL Mk. I, SAP Mk. I, Autocode etc. should be used, as specified in Vol. 2, Parts 1 and 2 of the 503 Manual.

The configuration for which the software system is most suited is one comprising:

At least 2 magnetic tape handlers

At least 1 unit of 16,384-word core backing store

1 line printer

If a wide range of equipment is available the system can take full advantage of this, yet will work quite adequately on a smaller configuration, provided that some form of auxiliary store is included. The adaptability of the system will be useful whatever range of equipment is available. For example, if a computer is fitted with tape handlers, one of them is used to hold the systems programs themselves. However, if the programmer wishes to reserve all the tape handlers for his own use, he may use the system, retaining only the utility routines which assist in the use of the tape equipment. In this way the system need not intrude upon the requirements of the programmer.

4. THE OPERATING SYSTEM

The operating system falls into two parts so that it can cater for all configurations of equipment and all types of programs to be run. The essential part of the system is the Executive program and this is always used whatever the configuration. It performs functions very similar to the Reserved Area Program used on the basic 503: it reads in programs, accepts messages from the typewriter, transfers control to the specified program, prints out messages etc. Using only the Executive, programs are run in the normal way as single units requiring the operator's intervention to initiate each program.

However, if magnetic tape handlers are available the Executive is supplemented by the Segmented Tape Administrative Routines (STAR), and used in conjunction with them. The systems programs are set up by the Executive and control is then passed over to STAR. All programs to be run under the STAR system are held on magnetic tape and are then processed automatically in accordance with the STAR commands tape.

The usual method of running programs under STAR is to group together several programs in a batch sharing a singe reel of magnetic tape. When a reel with several programs has been mounted on one tape handler, STAR ensures that any operations to be carried out for each program on the tape are executed automatically without manual intervention of any kind. Several operations can be carried out for each program in one run through the batch tape. For example, STAR can supervise the editing, retranslation and running of one program before passing on to the next, or the same program can be run more than once using different sets of data. This does not involve any restrictions on the individual programs of the batch, each of which can be designed and written as though it were going to be held by itself on its own separate reel of tape.

The input and preparation of a batch of programs onto a single reel of magnetic tape normally requires the supervision of an operator. However, the whole process can be speeded up by recognising that a high proportion of the programs to be placed in a given batch will have featured already in some previous batch, and can therefore be copied across directly from magnetic tape instead of being reinput from paper tape every time. For each installation there will probably be certain programs and subroutines of such frequent use that they can be copied among every batch of programs produced, thus saving a considerable amount of paper tape input.

Even more time can be saved on programs undergoing development. In this case, the magnetic tape holds not only the machine-code programs but also the source programs in ALGOL, SAP or FORTRAN, and in addition one or more sets of test data. Facilities are provided for making corrections to source programs by means of simple editing instructions which are obeyed when the program is copied across from a previous batch to the current batch; any necessary recompilation of the object program can be requested by the programmer and he may also call for simultaneous listings on the line printer. The programmer has, in fact, complete control over the content of his own segment of the batch tape, and he may insert, delete, and amend any of the items at will.

The batch processing methods provide an efficient way of using a computer especially in the case of programs lasting for a few minutes of running time. However, it is not always efficient or even possible to run programs in a batch. Large programs with a long running time are normally held on a separate reel of magnetic tape and run as a separate unit. For very short programs, lasting less than a minute, the batch processing has, in fact, several disadvantages. Firstly, the time taken to write the program to tape and bring it back again may be nearly as long as the time it would take to execute the program directly. Secondly, the programmer may have to wait several hours for other programs of the batch to be run before getting his results back.

For this reason, the method of 'load and go' is provided as an alternative to batch processing and may be used for short programs. This method can also be expedient in the case of programs of any length which are to be executed only once on any given occasion, and which are not therefore worthwhile putting on a batch tape.

A useful feature of the operating procedures is the possibility of interleaving batch processing with load and go operation. During the run of a batch the operator may interrupt at any time to request that at the end of the current STAR command the automatic sequence of processing should be temporarily suspended whilst he takes manual control. Under manual control, one or more load and go programs may be introduced before returning to automatic sequencing. This makes it possible to start running a batch of programs, even if it is known that the run will have to be interrupted in order, for example, to run a set of tutorial programs or for a short checking run of an urgent program.

Normally, in order to save excessive tape movement, the programs are executed in the order in which they appear on the batch tape. However, it is also possible to arrange for programs to be executed out of order, by setting the STAR priority list in the required order.

5. THE TIME-SHARING SYSTEM

The aim of the time-sharing system provided by the Peripheral Control Program (PCP) is to ensure maximum utilisation of the peripheral devices by keeping them occupied for as much of the run as possible, whilst the central processor, after initiating the peripheral transfers, is dealing concurrently with useful computation. This computation can belong to a program entirely independent of that which is using the peripheral devices, or it can belong to the same program; both possibilities are catered for by the time-sharing system.

The method of obtaining time-sharing within a single program is to regard the input and output routines as semi-autonomous programs which can run concurrently with the main program; thus once the initial order has been given, the main program can continue running at the same time as the input and output routines are waiting for the completion of a peripheral transfer. When a program requests a peripheral transfer, that request is put on a queue for the specified device and control returned immediately to the main program. If the device is free, the transfer is initiated immediately; if, however, the device is already busy, the request remains in the queue until its turn comes. This means that the main program is not held up and that work is available for a peripheral device immediately it becomes free.

The data to be transferred is input into buffer areas or output from buffer areas. Thus the system builds up queues of requests for buffers to be filled or to be output as appropriate. The administration of the scheme is carried out quite automatically and places no restrictions on the user, who may write programs as if the time-sharing methods were not being used.

The scheme for time-sharing within a single program will give excellent results in the case of programs in which there is sufficient computation to fill in all, or nearly all, the time occupied by input and output routines. However, there are programs which consist almost exclusively of input and/or output, and which contain hardly any calculation to fill the gaps; these are known as background programs. Since it would be uneconomic to run such programs by themselves, means are provided for running several background programs independently yet concurrently with the main (foreground) program. Provided that no two programs make use of the same peripheral devices, there is no limit to the number of background programs which may be run at the same time as each other. However, only one foreground program may be time-shared with the background programs at any one time.

When independent programs are time-shared, most of the central processor time is claimed by the foreground program, with only very short periods given over to background programs in order that they may initiate further peripheral transfers. The background programs can build up queues of requests for input and output devices in the same way as time-sharing within a single program.

Some restrictions are involved in the extent to which background programs may use other programs in the system; in particular they may not use the storage control system, and must be input by the load and go method.

Programs to be time-shared can be input to the computer at the end of an operation specified on the STAR commands tape. Once such a program has been entered it will proceed independently of all other programs being run simultaneously, until it signs itself off and informs the operator to that effect.

The time-sharing of independent programs greatly increases the range of tasks which can be performed economically on the computer. This includes such cases as:

1. Pseudo-off-line Conversion: the copying of data from an input medium to an output medium, with or without the incorporation of editing, checking and corrections. This covers the cases of transcription of data from paper to magnetic tape, from magnetic tape to line printer, to or from cards, and the control of the digital plotter by magnetic tape, paper tape, or punched cards.
2. Information scanning: a vast amount of data is held on magnetic tape, and a number of requests are made to retrieve items of information relevant to a small selection of subjects. This involves scanning each item of data held on the tape, to see whether it is relevant. If it is, it is retained, and if not, it is bypassed. The time taken to load and traverse the original file, which may be held on several reels, is time-shared with the normal work of the computer.

The system also provides the option of dynamic device selection by means of the Device Selection Program (DSP). When writing the program, the programmer need not specify the actual input and output devices he requires but simply refers to them as sources and destinations. At run time actual devices are assigned to each source and destination. This facility means that different runs of the same program can use different devices and, further, that the same program may be run on varying configurations of equipment without necessitating any alterations to the program.

If the programmer chooses to use dynamic device selection, he may then call on a further program in the time-sharing system, the Information Manipulation Program (IMP), in order to transfer data between his program and PCP. This program either accepts data from the object program and packs it into buffers for output, or unpacks data input into buffers and hands it over to the user program.

6. THE STORAGE CONTROL SYSTEM

The storage control system provided by the Storage Planning and Allocation program (SPAN) is designed to help the programmer make simple and efficient use of all forms of internal storage. The system allows fully dynamic allocation and reallocation of storage in blocks and permits the available storage to be used and re-used in the most economical fashion, according to the changing needs of the program. Note, however, that background programs may not use this facility; they must remain in the same area of store into which they are input.

6.1 Dynamic Storage Allocation

By using the storage control system, the programmer can allocate space as he requires it during a program and hand it back to the system for re-use (e.g. the buffer areas used in the time-sharing system). This means that the same storage area can be used over and over again for different purposes during the same run rather than being reserved exclusively for one purpose and usually being redundant for part of the run. Thus much larger tasks can be effected than would otherwise be possible.

6.2 Program Segmentation

Programs to be run under the storage control system are written in blocks (chapters), i.e. self-contained sections. Thus the system is able to administer programs which are too large to fit in the main store, by automatically placing blocks (chapters) of the program in core backing store and/or magnetic tape. A macro instruction is provided to jump to the next block of program required (or even a different program) no matter in which store it is held. If the relative addresses in the block need to be modified, this is done on transfer into main store. Under this scheme only those blocks which are actually being used at a given time need be in main storage, whilst the rest are held in auxiliary store.

6.3 Single Level Storage Simulation

The storage control system eliminates any problems presented by the different levels of storage when constructing general-purpose array-handling routines to operate indifferently on data held in any form of storage: by using the macros of the system the same instructions can be given to access the elements of arrays regardless of their position. In fact the programmer need not even know which form of storage is being used to hold the arrays, since the system can determine at any time the position and type of storage in which every block is held. Any necessary movement of data is effected automatically.

Thus, the programmer can regard all the available storage as a single level with immediate accessibility; in SAC he uses the macros to achieve this whereas in ALGOL and FORTRAN these facilities are incorporated automatically when a program is translated.

6.4 Direct Control of Storage

If he wishes, the programmer can gain a more direct control over storage allocation. He may state when requesting space which form of storage is required. Further, at any time he may specify that a block of program or data is to be moved to the most convenient form of storage; he may have a block deleted from main store, written to backing store and brought back when required; alternatively it can be deleted from main store and written to magnetic tape (banished). The macro 'retire' is a refinement of banish in that the information is not written to tape until the space in main store is actually needed for another purpose. If the space is not eventually required then no time need be spent in transferring the information between tape and main store.

The methods used by the storage control system are based on those described by I. K. Iliffe & J. G. Jodeit in their article 'Dynamic Storage Allocation' (Computer Journal, Volume 5, Number 3 pp 200-209, October 1962). The methods have a number of useful side-effects, of which full advantage is taken:

1. References to multi-dimensional arrays in symbolic programming languages are much faster by the SPAN method of "chaining" through the dimensions than with the normal method of address calculation.
2. Non-rectangular arrays (e.g. triangular and band matrices) can be compactly represented in store and efficiently processed.
3. It is possible to extend the size of a block of store if it turns out to be too small to hold the amount of information required.
4. It is possible to offset the range of indices so that the first element is referred to by an index other than unity; this is essential for the implemention of ALGOL.
5. During a checking run, the system will verify that every index used is within the permitted range, so that any attempt to corrupt the program itself or its constants or data, or any other program in store, can be detected and prevented.

7. THE PROGRAMMING LANGUAGES

The three programming languages in the system are Symbolic Assembly Code (SAC), ALGOL and FORTRAN. The translators for these languages, the Symbolic Assembly Program Mk.2 (SAP), the Elliott ALGOL Mk.2 Compiler and the FORTRAN IV Compiler, produce programs to be run in conjunction with all the other systems provided. Programs written in FORTRAN and ALGOL automatically use the facilities of the time-sharing and storage control system. In SAC the functions of these systems can be invoked by the use of built-in macros.

A most useful feature of the languages is their flexibility: for example, a program written in either ALGOL or SAC can be called as a slave program by a master program in the other language. Further, sections of SAC can be included in an Elliott ALGOL program. The FORTRAN language has the advantage of independent compilation of each unit making up the complete program, i.e. master program, slave programs, sub-routines and standard function routines can all be translated separately and later be incorporated into a complete program.

All translators produce their object program in a single pass, and allow the program to be run immediately after input of the source program, on the load and go principle. The speed of translation is such that the source program is input at nearly the full speed of the input tape reader, so that no efficiency is lost by load and go methods; however, the translators also operate in conjunction with the batch processing system for programs which are more conveniently run in that manner. In this case the object program is compiled and then output in binary form to paper or magnetic tape. It can then be reinput for batch processing without being translated each time.

The SAC language is provided for the programming of jobs which require close control of the actual coding produced. The language includes all the normal facilities of symbolic assembly in addition to powerful macro-compiling capability.

The ALGOL translator is designed in accordance with the recommendations of the European Computer Manufacturers Association. These recommendations are based on a full subset of ALGOL, which includes all facilities which are ever likely to be found useful in practical programming.

The ECMA subset is sufficiently full to include all the facilities of the subset specified by the International Federation for Information Processing, the official body for the maintenance of ALGOL. It is expected that most published programs will conform to this smaller subset, and therefore will automatically be acceptable to translators which follow the ECMA recommendations.

On the 503 even some of the restrictions of the ECMA subset have been relaxed; in particular, **own** variables are permitted and most of the forms of recursion are allowed. These relaxations are made without impairing the efficiency of the object programs which do not use them. Programs written in Elliott ALGOL Mark 1 and in 803 Elliott ALGOL are also acceptable provided that they do not use certain non-ALGOL optimisation features; in particular, calls of the 'elliott' order are not permitted.

The 503 FORTRAN IV Translator has also been designed according to ECMA recommendations with certain extensions and relaxations. The American Standards Association definition of FORTRAN IV was published in the Communications of the A.C.M., October, 1964. ECMA FORTRAN is a subset of this, the major differences being that it excludes the use of complex and double length data, the functions for complex and double length data and data initialisation statements. The translation speed and the efficiency of the object program are approximately the same as for ALGOL Mark 2.

8. THE PROGRAM CHECKING SYSTEM

The first aim of the program checking system is to make a check of the validity of the source program. This excludes the possibility that a slightly wrongly phrased source program will produce a wildly inappropriate object program, which will be practically impossible to check. When an error has been detected during program input, the diagnostic information is conveyed to the programmer, and the process of checking continues, although translation is stopped and the program will never reach the running stage.

Another function of the checking system is to make extra tests of validity during the running of the program under review, which would not be desirable during production runs because of the extra time involved. The most important of these is the check on the values of indices (subscripts) which prevents a program in the store from corrupting itself or anything else. When an error is detected, its nature is indicated, together with as much additional information as is requested by the programmer. Then, at the programmer's option, the run will continue.

When the programmer has traced his error to the source, the program must be corrected before it is run again. The checking system provides a convenient way of doing this: instructions for the insertion, deletion and replacement of portions of the text of the program are punched on a short length of tape, which is input to the computer ahead of the program to which it refers. The checking system ensures that only the characters of the corrected text reach the compiler. In order to verify that the corrections have been correctly specified, a complete listing of the corrected text on the line printer may be requested; when the program is working, a fresh copy on paper tape may also be requested. Although most programs are part of a batch, there is no restriction on the number of operations which may be carried out in connection with one program before passing to the next.

The checking system includes the normal facilities for optional printing of the values of intermediate results, and for tracing through the course of the program. The operations of the system are similar for all the languages and their use is integrated with the process of translation. Communication between the computer and the programmer is exclusively in the source languages, and variables and labels are always referred to by the symbolic names given to them by the programmer, not by the addresses allocated to them by the computer. This secures a more efficient use of the computer, since the printing out of storage maps and octal listings can be completely avoided.

When magnetic tape handlers are available all checking and diagnostic information is written to magnetic tape; at the end of the checking run this tape is rewound, and the information it contains is transcribed to the line printer. This method has the advantages that:—

1. The line printer is left free to accept the normal output of the program.
2. It is possible to omit the printing up of the early checking information, produced by the correctly working part of the program, and concentrate on that which was produced immediately prior to the occurrence of an error.

However, if no magnetic tape is available or the programmer has already reserved all the handlers, the checking information can be output to paper tape or printed on the line printer as it becomes available.

The operating system enables checking runs to be loaded in between production runs, so that the programmer with a program undergoing checkout can gain frequent access to the computer and his results are available immediately after the run.

503 TECHNICAL MANUAL

VOLUME 2 : PROGRAMMING INFORMATION

PART 4 : THE PROGRAMMING LANGUAGES FOR THE NON-BASIC 503

SECTION 1 : SYMBOLIC ASSEMBLY MARK 2

SYMBOLIC ASSEMBLY MARK 2

PREFACE

The reader is assumed to be familiar with Section 2.1.1. of the 503 Computer Manual, which describes among other things, the number representation and function code of the 503: also a knowledge of the conventions of 803 T.I. Code is assumed.

The Symbolic Assembly Program Mark 2, in future referred to as SAP 2, is part of a complete software system for programming large 503 computers. This software system includes routines for storage planning and allocation (SPAN) and for control of the 503 peripherals (PCP). These routines provide many facilities which in effect extend the hardware functions of the computer.

This description of SAP 2 is divided into three subsections. The first five chapters explain the use of Symbolic Assembly Code (SAC) and give a short introduction to SPAN and PCP. The next ten chapters, numbered 6 to 15 give instructions on how to write code in a form which is acceptable to SAP 2. Providing the conventions given in these chapters are adhered to, the programmer need not concern himself with any restrictions imposed by the use of the routines of the software system. Chapters 1 to 15 are intended as a guide to programming in SAP 2. The later chapters contain further details of the facilities of SAP 2 and give the operating instructions for assembling and running SAP 2 programs.

CONTENTS LIST

CONTENTS LIST (Continued)

CONTENTS LIST (Continued)

CHAPTER 1

SYMBOLIC ASSEMBLY CODE

1.1 Reasons for Symbolic Code

Most instructions on the 503 consist of two parts, a function part and an address part (see 2.1.1). In use, each of these parts consists of a number, the function part being the numerical code of the function required, and the address part being, in general, the number of the location which contains the operand.

In the case of the address, the user is not really concerned with which storage location is in use – any will do, provided the choice is consistent.

The basic purpose of the Symbolic Assembly Code is to ease the difficulty of addressing. This is achieved in the following way. Any location may be given a name, or *identifier* and be referred to by that name. The name is usually chosen to be meaningful, like "speed" or "time", whereas the number of the storage location is unlikely to be significant or memorable. These names are chosen by the user.

The basic function code of the 503 is, in the main, consistent and reasonably easy to memorise. However, the Symbolic Assembly Code enables the programmer to use certain groups of letters to represent the basic function code, and, in effect to extend the basic code by using a group of letters to represent a sequence of functions. These groups of letters are called *mnemonics.*

1.2 Compiling

Programs written in Symbolic Assembly Code (SAC) are not in a form which can be obeyed by the computer. Before the program can be placed in the store and obeyed it must be translated into numeric form, and all the decisions such as which data to place in which store location must be taken. The resulting program is called the *object program.*

The task of translation from symbolic code to object program is purely routine, requiring only persistent application of rules. In fact the task can be undertaken by a computer and is then usually called compiling. The program which does the job is called a compiler.

The compiler which converts the Symbolic Assembly Code (SAC) to Object Program is called the Symbolic Assembly Program MK 2 and will be referred to as SAP 2.

When a SAC program has been written it is punched on to paper tape, SAP 2 is loaded into the computer and it reads the S.A. Code either directly from paper tape or from magnetic tape.

As the SAC program is translated the object program is placed in the main and auxiliary store of the computer. Subsequently the object program can be run or it can be output to paper tape or magnetic tape for running at a later date.

CHAPTER 2

STRUCTURE OF SYMBOLIC ASSEMBLY GROUPS

A SAC program may be regarded as a string of characters which are capable of being compiled into an object program. The characters are selected from upper and lower case letters, the digits and a variety of signs and punctuation marks.

The string of characters is divided into *groups*, each group being normally written on a different line. Most groups correspond to instructions in the desired object programs. The remaining groups are used to control the operation of SAP 2 and are known as *controls*. Controls are used to indicate natural divisions of the program, specify storage requirements for data, and so forth.

The groups which correspond to instructions of the object program on a one to one basis are known as *instructions*. Those which correspond to a one to many basis are known as *macros*.

The group of characters is further split up into *items*. The most important types of item are as follows:

Identifiers are invented by the user and consist of a string of letters and digits beginning with a letter. They may be of any length but only the first six characters are considered by the compiler, any extra ones being ignored. An identifier may not be followed immediately by a further letter or digit since this would merely be accepted as a continuation of the identifier.

Numbers consist of a string of decimal digits which have the usual significance of specifying an integer.

Mnemonics are strings of characters having pre-assigned meanings in SAC. They may be of any length but only the first six characters are considered by the compiler.

The difference between a mnemonic and a macro is shown by the following example:

enter [B, n]

The complete group is a macro, enter is the mnemonic part and the items B and n are parameters. Thus enter , has a pre-assigned meaning and B and n are identifiers which further specify the series of instructions represented by the macro.

Octal functions are a pair of octal digits which represent the function in numeric form.

In the case of instruction groups the first item is normally an octal function or a mnemonic which specifies the function part of the resultant instruction in the object program. This function part may be preceded by one or more identifiers, each followed by a closing round bracket. An identifier which is introduced in this way is called a *label*, and serves to identify the group which it precedes so that reference may be made to that group from elsewhere in the program.

The first item of a macro is always a mnemonic which specifies the function of a set of instructions in the object program. This mnemonic may be preceded by one or more labels in the same way as an instruction group.

Control groups may not be preceded by labels.

CHAPTER 3

PROGRAM STRUCTURE

3.1 Chapters and Blocks

A SAC program is not normally written as a single unit, but is split up into blocks.

The division into blocks is merely a convenience for the user. It is usual to consider the program as separate sections, since most programmers find that a section containing a few hundred instructions is as large as can comfortably be worked on at one time. The division into blocks is mainly a programming convenience, but has some effect on the object program. For example, a new block always begins in a new location, even if a half-location is spare at the end of the previous block. Also, it is not allowable to proceed "off the end" of one block and thus into the next. Blocks must always be entered and left by jump instructions.

When a program has been written as a series of blocks, the blocks may be grouped into one or more *chapters.* A small program will have only one chapter but a large program will usually be divided into several chapters so that the ones which are temporarily not in use can be held in auxiliary store, e.g. core backing store or magnetic tape.

A chapter normally occupies a number of consecutive locations of the store. The important feature of a chapter is that it may be bodily copied into any set of consecutive locations of the main store, and may be used in that position. That is, a chapter is relocatable, its position in store does not matter.

In order that chapters may be relocatable certain conventions must be observed. SAC is designed to observe these conventions and the inexperienced programmer has only to follow the rules. Facilities are available for the experienced programmer to relax some of the rules but this must be done with great care.

3.2 Dynamic Relocation

It was stated in the last paragraph that chapters could be copied into any set of consecutive locations in the main store and be obeyed there. This process is dynamic relocation. It is not the same as the relocatable input available on the 803 or the RAP relocatable binary input of the 503, where the ~~destination~~ of the program is decided on input, and cannot be altered without re-input | [illegible] position

Dynamic relocation is provided by the automatic use of SPAN by SAP 2. Thus a single chapter of the SAC program is compiled by SAP 2 into a single SPAN block which contains not only the object code but also relocation markers which record which addresses have to be altered if the chapter is moved in the store.

In general programs should not be written so that instructions are altered at run time. If they are, the relocation markers, which are formed at compile time, may not correspond to the relocation required by the altered instructions.

3.3 Identifier Introductions

Identifiers are introduced either by attaching them to instructions as labels as described in Chapter 2 or by use of a control group. The control group consists of a control name followed by a list of the identifiers. Thus, for example, if a program contains blocks which are identified by alpha and beta and uses the data locations time and space, the following introductions must be given

```
block  alpha, beta;
data   time, space;
```

3.3.1 Scope of Identifiers

An identifier introduced within a block is said to be *local* and only has meaning within the block. An identifier which is introduced outside a block is said to be *global* and has meaning anywhere within the program.

In addition certain other identifiers, such as the one which names the program itself, have meanings outside the program and are called *universal*. (see P9)

No two identifiers with the same scope may be identical. However a local identifier may be identical to a global identifier, and two local identifiers can be identical provided they are introduced within different blocks.

3.3.2 Use of Identifiers

Identifiers are used within a SAC program to name blocks of program, label instructions within a block and to identify data locations.

Block and data identifiers must always be introduced before they are used in an instruction.

It must always be possible to refer to a block anywhere within a program and therefore block identifiers must be global. Data and label identifiers however can be either local or global.

A local label is introduced by attaching it to the relevant instruction. If however, it is required to use a label globally then it must be introduced at the same time as the block within which it is introduced locally.

Outside its own block the instruction it labels must always be referred to by the label identifier and the block identifier.

3.4 Macros

It was mentioned at the beginning of Chapter 2 that a group of characters which corresponds to instructions on a one to many basis is called a macro.

A macro consists of a mnemonic which may be followed by a list of identifiers enclosed in square brackets.

A macro is written on a line by itself and during compiling is replaced by a set of machine instructions.

A data or label identifier is distinguished from an identical macro mnemonic by the compiler.

CHAPTER 4

STORAGE PLANNING AND ALLOCATION IN SAP 2

In Chapter 3.2 it was mentioned that relocation was implemented in SAC programs by the use of the Storage Planning and Allocation Routines (SPAN). These routines are fully described in Section 2.5.2. of the 503 Manual. The purpose of the notes in this section is to indicate to the programmer how SAP 2 uses SPAN to handle his object program and how he may use SPAN within his program.

Under the control of SPAN, the core storage of the machine is divided into blocks of consecutive words. Additional blocks may be held on magnetic tape. The first word of each SPAN block, (called the *header word)* describes the block, giving its length and the type of information held in it (e.g. data, program, free space). Each block has also a codeword which is effectively a pointer to the position of the block in whichever form of storage it has been placed. The system keeps in fixed workspace sufficient codewords to be able to determine at any moment the whereabouts of every SPAN block. When a program makes a request for more space than is available in the main store, SPAN moves some blocks into auxiliary storage in order to make room. If the programmer then makes reference, via SPAN, to a program block which has been moved into auxiliary store, the block is brought back into main store, moving out other blocks if necessary to make room. Similarly if reference is made to an element of a data block on magnetic tape, the block is brought back into main store. This movement of information takes place automatically.

4.1 Use of SPAN by the SAP 2 compiler

When a SAC program is compiled by SAP 2 a SPAN block is allocated to each chapter of the SAC program. In addition SPAN blocks are allocated to hold the codewords of these blocks and to store the workspace of the chapters.

4.2 Use of SPAN by the SAC Programmer

The most common use of SPAN by the SAC programmer is for dealing with *arrays* (e.g. matrix manipulation).

When writing in 803 T.I. code it is always necessary to allocate storage space at *compile time* for the maximum size array that will be used.

The SAC programmer can, however, by the use of SPAN, allocate storage space *dynamically* so that he only uses sufficient storage for his needs in that run of the program.

For a single dimensional array there is a single codeword pointing to a data block which holds the array elements. For a two dimensional array the main codeword points to a block of codewords each of which points to the elements of one row of the array. As a result, an array element in core store can be accessed by "chaining" through the codewords. This system means that it is not necessary to have the same number of elements in each row of an array.

The SAC programmer is provided with facilities for allocating store and accessing elements by the use of macros which are recognised by the SAP 2 compiler.

CHAPTER 5

INTRODUCTION TO INPUT/OUTPUT PROGRAMS

The SAC programmer must use PCP to control all input and output, either directly, via the PCP macros provided, or indirectly, by use of IMP and DIOR. This chapter gives an introduction to each program.

5.1 The Peripheral Control Program (PCP)

PCP ensures maximum utilisation of the peripheral devices. It makes full use of the Autonomous Data Transfer (ADT) and Interrupt facilities of the 503 and provides a time sharing system.

PCP makes it possible for the programmer concerned with the main program to make requests for serveral devices to transfer information concurrently into or out of the 503 while the main program is operating.

5.1.1 Buffers

PCP carries out all data transfers to and from the 503 via buffers. A buffer is an area in main store used by PCP to store incoming or outgoing data. The programmer would normally obtain a buffer either directly from SPAN or through PCP, although it is possible for the programmer to use his own workspace. Care must be taken to ensure that the buffer is in a fixed position and is specified by the address of its codeword when a request for a data transfer is made by a PCP macro.

The PCP input/output macros include parameters to specify the device to be used and name the buffer to which the request applies.

5.1.2 Queueing

When a transfer request is made, PCP attaches information relating to the buffer and the function to be performed to a queue for that device. If the device is free the operation is performed immediately; if not, information is left on the queue until the device becomes free.

In either case the transfer is carried out autonomously and control returns immediately to the calling program. This prevents the main program being held up and ensures that the peripherals are used efficiently.

If the calling program gives a specific instruction that it is not to be resumed until the transfer operation is completed, another program is entered. If no other program is currently in operation then the system waits until the operation is completed.

5.1.3 Device Routines

The SAC programmer using PCP has the additional advantages provided by the Device Routines which check the accurate functioning of all devices, where possible, and which contain instructions to repeat operations in which a parity or similar fault occurred.

5.1.4 Control

The system provides control for all 503 input/output devices e.g.

Paper tape input
Paper tape output
Magnetic tape
Magnetic film
Digital plotter
Line printer
Control typewriter

A full description of PCP is given in Section 2.5.3 of the 503 Manual.

5.2 The Device Selection Program (DSP)

The Device Selection Program is a set routines used in conjunction with PCP which permits the SAC programmer to refer to input/output devices by a formal numbering system. The actual devices corresponding to the formal numbers, may be specified at the end of the program but do not have to be specified until run time. This means that the same program can be run on different 503 configurations, or on the same one but using different peripherals, without alteration.

5.3 The Information Manipulation Program (IMP)

IMP is a set of routines used in conjunction with PCP and DSP which enables the program to input/output single items of data instead of dealing with complete buffers. On output, IMP forms buffers from the individual items of data and outputs the information to the selected device using DSP and PCP. On input, IMP inputs complete buffers or information and transmits individual items to the programmer on request.

5.4 Data Input/Output Routines (DIOR)

Routines are provided for reading and printing fixed and floating point numbers and strings. These routines make use of the facilities of PCP, DSP and IMP.

CHAPTER 6

BLOCKS

6.1 begin

It was mentioned earlier that a program is not normally written as one unit but is split into sections of program called blocks.

A block must be named so that it can be referred to by instructions within other blocks. The compiler must also be informed that this is the beginning of a block. Therefore at the beginning of every block the programmer writes the control name begin followed by the identifier which names the block, followed by a semicolon. (Note that control names are always underlined).

In general the block identifier is chosen to give some indication of the result of obeying the block of instructions.

e.g. begin root;

6.2 Local Data

The declaration at the beginning of the block is in general followed by the introduction of the local data identifiers. This is done by writing the control name data followed by the list of local identifiers, separated by commas followed by a semicolon.

e.g. data numbers, answer;

Then follow the actual program instructions, one instruction to a line. Most instructions consist of a function and an operand.

6.3 Functions

A complete list of the basic machine functions is given in Chapter 2.1.1.1. of the 503 Programming Manual. They can be represented by a pair of octal digits

6.4 Operands

The operand is either a literal e.g. 51 5 ,or is a location of the store. Within a SAC block a location of store may be referenced either by an identifier or by use of the "diamond bracket" facility, which is fully described in paragraph 6.9.

Thus one can write

```
30 number
```

where number is a data identifier or

```
40 again
```

where again is a local label attached to some instruction within the block.

Alternatively one can write

30 < + 2 >

which means "set the accumulator equal to the value 2".

6.5 Local Labels

It was stated in Chapter 3 that a local label identifier is introduced by prefixing it to the instruction which it is intended to identify. This is done by preceding the instruction by the identifier followed by a closing round bracket.

Each instruction may be given up to 3 labels, e.g.

L 1) again) repeat) 30 a

These labels may be used in the address part of any instruction within the block in which they are introduced, and refer to the instruction and not the instruction pair which follows them.

6.5.1 Jump Instructions

Whenever the programmer wishes to transfer control to a labelled instruction within the same block, he uses the functions 40, 41, 42, 43. The SAP 2 compiler will automatically convert these functions to 44, 45, 46, 47 if the labelled instruction is placed in the second half of the word in the object program.

6.5.2 Dynamic Stops

No SAC program should contain dynamic stops. These are replaced in the 503 system by the following mnemonics:

STOP	– unconditional
OSTOP	– stop if overflow set
NSTOP	– stop if the accumulator is negative
ZSTOP	– stop if the accumulator is zero.

When the program is being run these mnemonics have the effect of transferring control to the overall operating system.

6.6 End of Block

When all the instructions of a block have been written the programmer must write the control name <u>end</u> followed by a semicolon to indicate to the compiler that the block is complete. The programmer may insert any characters between <u>end</u> and semicolon as they will be ignored by the compiler, though the first of these should, of course, not be the underline character.

6.7 Example

The following example illustrates the previous points. It is a re-write in SAC form of the example given in Chapter 2.1.1.1. of the 503 Computer Manual.

To extract the positive square root of a fraction the formula

$$y_{u+1} = \tfrac{1}{2}\left(y_u + \frac{x}{y_u}\right)$$

is used to generate a series of numbers y_i which converge monotonically to the root of x.

The initial value of y is taken as $1-2^{-38}$ as this is the largest positive number which can be held. Ideally $y_{u+1}-y_u$ should eventually reach zero but rounding errors can cause oscillation between plus and minus 2^{-38}. The process is therefore repeated until $y_{u+1}-y_u$ is zero or positive.

It is assumed that the number is in the accumulator at the start. The root is in the accumulator "on exit from the block".

```
                                              comments

begin     root;
data      number,yu;
          NSTOP                           stop if negative
          42   end                        exit if zero
          20   number                     store the number
          30   <37 8191/77 8191>          } store initial value of y u
          20   yu                         }
again)    30   number                     form 2y u+1 =( y u + x/y u )
          56   yu
          04   yu
          51   1
          15   yu                         Write y u+1 , form y u+1 - y u
          41   again                      If negative repeat
          07   yu                         If positive exit with yu.
end)      exit [1]
end root;
```

Note: The explanation of the instruction exit [1] is given in Chapter 7.4.

6.8 Wholewords

So far this chapter has dealt mainly with instructions which are written, one to a line and which in general occupy half a computer word. Other information can be written in a SAC program which defines the content of a complete computer word. Any SAC expression which represents the content of a complete computer word is called a *wholeword.*

The following types of wholewords are described below.
- Integers, Fractions and Floating Point numbers,
- Octal Groups.
- Instruction pairs.

6.8.1 Integers, Fractions and Floating Point Numbers

These are composed of the digits 0 to 9 and the characters +− and subscript $_{10}$. Each number must contain at least one digit and be preceded by a sign.

Any whole number between −274 877 906 943 and + 274 877 906 943 inclusive is called an INTEGER

e.g. +5 −304882

A FRACTION consists of the character . (point) followed by no more than 11 digits: it may be unsigned.

e.g. – . 11129811392 .5 +.5

Note that signed fractions such as +0.113 will also be accepted by SAP. Unsigned fractions of the type 0. 113 should not be used.

A FLOATING POINT NUMBER is one of the form

$$a \times 10^{b}$$

'a' being known as the 'mantissa', and 'b' as the 'exponent'.

The form of the mantissa is an integer, or a fraction, or an integer followed by an unsigned fraction. It must be such that the number obtained by removing the character 'point' (if present) is an integer as defined here.

The exponent is an integer of no more than 2 digits, and may be without a sign if positive, or even omitted altogether if zero.

A floating point number is written as

mantissa_{10} exponent

and floating point overflow will occur if it is outside the (approximate) range:

$\pm 1.73 \times 10^{-77}$ to $\pm 5.8 \times 10^{76}$

Examples $+3_{10}+5$ (=300,000)

$-7.33192_{10}+60$

$+.892_{10}-27$

$.5_{10}$

$+0.739_{10}\ 2$

Numbers beginning with character $_{10}$ will also be accepted, and are understood to have a mantissa of 1.

Example: Zero may be expressed as

+0 –0.0 .0 $+0_{10}$ +00000 and so on.

But '.000000000000' is erroneous, because it is a fraction with more than 11 digits, and so is '+' because it holds no digit.

6.8.2 Octal Groups

A 39 bit location content may be split into 13 equal parts of 3 bits each. If each of the parts so obtained is expressed as a digit between 0 and 7, and the whole expression is preceded by the digit 8, the result is known as an 'octal group'.

For example the number 21 may be written as

80 000 000 000 025

in octal form, since 21 may be expressed as the 39 bit location content

```
        . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .1.1.1
        –21 is87 777 777 777 753
```

6.8.3. Instruction Pairs

Instructions in SAC programs are normally written one to a line and the compiler packs them two to a word when forming the object program.

The SAC programmer may, however, inform the compiler that he requires an instruction pair to occupy a whole word.

He may do so in any one of the following ways:-

a) Write two instructions on a line, separated by the B digit indicator colon (:) or slash (/). This method is often used to express a constant as a pseudo order pair.

```
Example + 100  is equivalent to  00   0  :  00 100
    and – 100  is equivalent to  77 8191/77  8092
```

b) Write the instructions on separate lines but terminate the first instruction of the pair by a colon or slash.

```
e.g,    22 number /
        30   P
```

c) Precede an instruction by a colon. This indicates to the compiler that this instruction is to occupy the second half of a wholeword the first half of which is zero

```
e.g.                : 20 B
is identical to  00 0 :  20 B
```

```
Note that        00 0 : 00 P
                      : 00 P
                        +P
```

All have the same effect. i.e. the location contains the address of the identifier P. The identifier P can be a data identifier or a *local* label identifier.

6.9 Diamond Bracket Constants

Wholewords may be stored in a SAC program by writing them as part of the program with label for identification.

```
e.g.  two)  +2
```

This constant may then be referred to in an instruction such as

```
30  two
```

the result of which is to place +2 in the accumulator.

An alternative way of doing this is by the instruction

```
                30 < +2 >
```

The SAP 2 compiler allocates storage space to wholewords read in this way. Such wholewords are stored in a separate SPAN block and are available throughout the program. They are not duplicated if read again within the same program.

One may write any form of wholeword within diamond brackets with the exception of diamond brackets themselves.

Thus one may write

```
                    30 < : 30 P >
```

but one *cannot* write

```
                    30 < : 30 <+6> >
```

This must be written either as :

```
                    30 label
                      .
                      .
                      .
        label)      : 30 <+6>

  or                30 <: 30 L>
                      .
                      .
                      .
          L)        + 6
```

Note that one can write

```
                    30 <+P>
```

where P is a data identifier or a *local* label identifier.

6.10 Comments

Comments in a SAC program which are helpful to the programmer may be punched on the tape if desired without affecting the translation of the program, provided they are enclosed within round brackets. e.g. 30 A (A = collating constant)
is read as 30 A
by the SAP Mk 2 compiler. There is, however, one restriction that comments must *not* be punched inside any introduction list.

CHAPTER 7

SIMPLE ENTRY AND EXIT FROM BLOCKS

When entering a SAC block it is necessary to name the block which is to be entered and the label of the first instruction to be obeyed. Also one must use a *transfer* macro instead of a group 4 instruction to enable SPAN to ensure that the block required is in main store.

There is a convention which states that the block identifier also labels the first instruction of a block. For example consider the block below:

```
begin sum;
data  all, modsum;
      24  all
mod)  01  0
      41  mod
      24  modsum
      .   .
      .   .
      .   .
end   sum;
```

if in another block one wrote

trans [sum*sum]

then the effect would be to enter the block at the first instruction and so add the contents to the location all and its modules to location modsum. However, writing:

trans [mod*sum]

would cause an entry at the location labelled mod and thus an addition to modsum only.

Note that sum*sum and mod*sum are relative identifiers. A relative identifier has the form A*B where B is a blockname and A is either a label or data identifier.

7.1 Global Label Identifiers

In Section 2.5 a description was given of the introduction of local label identifiers and it was stated that these identifiers only had meaning within the block in which they were introduced. In the above example a label is used outside its own block. Therefore it is being used as a global label and so must be introduced.

Block identifiers and labels are introduced globally by giving a block introduction at the beginning of the program, this is done by writing the word block underlined, followed by the block identifier followed by the labels within the block which are to be used globally.

e.g. block sum (mod);
     block one (alpha, beta), two, three;

Note that block identifiers are separated by commas and that the label identifiers are contained within round brackets and are separated by commas. The last introduction is followed by a semicolon.

7.2 Transfer Macros

These are four transfer macros corresponding to the four Group 4 instructions:

```
 trans  [A*B]  unconditional
ntrans  [A*B]  negative
ztrans  [A*B]  zero
otrans  [A*B]  overflow
```

where B is a block identifier and A is a global label. For transfer to the first location of a block the global label A is replaced by the block identifier B.

```
e.g.  ztrans [B*B]
```

7.3 Global Data Identifiers

Local data identifiers were described in Chapter 6.2.

Global data identifiers can also be used and they are introduced in the same manner as local identifiers but externally to any block. They are usually given at the beginning of the program following the block introductions.

If the same identifier is introduced globally and locally then within the block in which it is introduced the local meaning is assumed.

For example

```
block   alpha,beta;
data    one;
begin   alpha;
          30  one
          51   1
          trans [beta*beta]
end     alpha;
begin   beta;
data    one;
          20  one
          STOP
end     beta;
```

An entry to block alpha would leave the global data one unchanged but would cause the local data one of block beta to be altered.

If after a block is written, including local data introductions, it is found that local data is required globally then a global introduction must be given in the form

```
data  one*beta;
```

Thus is the above example the global introductions would become

```
block  alpha, beta;
data   one, one*beta;
```

7.4 Subroutine Entry and Exit

The SAP 2 compiler automatically allocates a 'link-location' for each block it reads doing so at the time the block identifier is introduced.

Thus each SAC block may be treated as a subroutine provided that its exit instruction is correctly arranged.

When entering a SAC block as a subroutine it is necessary to use the macro instruction enter so that SPAN can ensure that the subroutine required is in main store.

The enter macro is written in the form

enter [B,n]
enter [A *B,n]

where B is the block identifier, A is the global label and n is the number of parameter words after the 'enter' macro.

If a SAC block is intended to be used as a subroutine then exit must be made by use of the "exit" macro which is written in the form

exit [m]

where exit is to the mth location after the entry macro. In general m = n + 1.

There are standard conventions for dealing with parameters of subroutines and these are explained in chapter 5.

Example

Given x and y to find $\sqrt{(x^2+y^2)}$ and store it in z. It is assumed that x and y are fractions.

```
block   square,root. . . . . . .;
data    x,y,z;
           :
begin   square;
data    workspace;
        30  x
        53  x
        20  workspace
        30  y
        53  y
        04  workspace
        enter [ root,0 ]
        20  z
           :
end     square;
begin   root;
data    number,yu;
        NSTOP                    (stop if negative)
        42  end                  (exit if zero)
        20  number               (store the number)
        30  <37 8191/77 8191>    (store initial value of y_u)
        20  yu
again)  30  number               (from 2y_u+1)
        56  yu
        04  yu
        51  1
        15  yu                   (Write y_u+1, form y_u+1 -y_u)
        41  again                (If negative repeat)
        07  yu                   (If positive exit with yu)
end)    exit [ 1 ]
end     root;
```

CHAPTER 8

ARRAY ALLOCATION

8.1 Data introductions

So far a description has been given of the use of identifiers to refer to single words of data.

Mention was made in Chapter 4.2 of the 803 T.I. method of allocating data space at compile time for arrays.

The method can still be used in a SAC program by writing introductions in the form

```
data  number (12)
```

This introduction causes 13 locations to be reserved and they may be referred to in the program as number, number + 1, number + 2, ... number + 12 or by the use of B-lined instructions.

Example:

```
         data   number (12), total;
         begin  sum;
         data   count;
                26  total         (clear total)
                30  <-12>
                20  count         (set count)
repeat)         00  count   /
                30  number +12  (pick up number)
                24  total         (add to total)
                32  count         (test & increment count)
                41  repeat
                exit [ 1 ]
```

Note that B-lined instructions are also written one instruction to a line. The stroke, at the end of the instruction labelled repeat, informs the compiler that this instruction must be stored in the next available first half word.

8.1.1 Compiling of data introductions

When, during compiling, a data introduction is met locations are reserved in *main store* and these identifiers are associated with *main* store locations throughout the running of the program.

Thus, while a program is running, main store locations are permanently reserved for all the data introductions.

Therefore the above form of data allocation should only be used for arrays of known length and is general, only for short arrays.

8.2 Array Introductions

In chapter 4.2 it was also mentioned that, by the use of SPAN, the SAC programmer can allocate storage space dynamically. This is done by use of the control word array and the SPAN allocation macro. (see Pg ! which may only be introduced universally.)

Thus the programmer introduces an array by writing an identifier after the control word <u>array</u>. Later in the program he asks SPAN to allocate a block of storage and then informs SPAN that the codeword of the block is to be stored in the location specified by the identifier.

8.3 Use of SPAN for dynamic allocation of storage

A full description of the facilities of SPAN is given in section 2.5.2 of the 503 Manual. The following paragraphs describe the structure of SPAN blocks and explain the use of the SPAN macros

```
ALLOC
consider
row
insert  delete
tk 1  st 1  tk 2  st 2
take  store
length
```

It was stated in Chapter 4 that under the control of SPAN, the core store of the machine is divided into blocks of consecutive words and that additional blocks may be held on magnetic tape..The first word of each SPAN block (called the header word) describes the block, giving its length and the type of information held in it. Each block also has a *codeword* which is effectively a pointer to the position of the block in whichever form of storage it has been placed.

Whenever a SAC programmer wishes to reserve storage dynamically he must ask SPAN for a block stating at the same time the size, the type of information to be stored in the block, the "storage status" of the block and the required "offset".

size:	The *total* number of locations in the block i.e. one location for each data item plus a further location for the header word.
type:	The type indicates whether the content of the block is numbers, program or codewords.
storage status:	The storage status tells SPAN whether the block is to be kept in main store or in backing store or whether SPAN can make the decision.
offset:	The offset has no other purpose or effect than to change the index range of the block so that the elements are no longer numbered from 1 to $l-1$, where l is the length of the block, but rather from $(-\text{offset}+1)$ to $(l-\text{offset}-1)$.

8.3.1 The SAC programmer requests storage from SPAN by use of the allocation macro. This consists of the mnemonic ALLOC followed by identifiers which name locations which contain information on the size, type, storage status and offset of the block required. The identifiers are separated by commas and are surrounded by square brackets.

e.g. ALLOC [Size, Type, Status, Offset].

Size:	contains a positive integer specifying the size of the block.
Type:	contains an integer specifying the contents of the block, e.g. +1 if block to contain data +2 if block to contain program +3 if block to contain codewords

If the SAC programmer requires the index to be checked, on each reference to the block, to see that it is within the range of the block then the integers +5, +6, +7 should be used in place of +1, +2, +3.

Status: contains an integer specifying the type of storage required for the block e.g.
+0 if the block can be in main store or backing store. This is the normal form.
+1 if the block is to be kept in main store.
+2 if the block is to be kept in backing store.

Offset: contains an integer equal to (1–lowest index). Thus if the contents of the block are to be referred to by the index numbers 1 to $l - 1$ then the value of the offset is zero. This is the usual case. (If a non-zero offset is required see Section 2.5.2 of the 503 Manual.)

Examples a) To request a SPAN block which is to contain 20 words of data which are to be indexed from 1 to 20. The block can be in either main store or backing store

```
ALLOC [ <+21>, <+1>, <+0>, <+0> ]
```

b) To request a SPAN block as above except that the number of words of data is stored in the location 'data'

```
02  data
20  temp
ALLOC [ temp, <+1>, <+0>, <+0> ]
```

Note that in ALLOC the diamond bracket form of specitying constants may be used.

8.4 Storage of Codeword

Having allocated a block of storage the SAC programmer must then tell SPAN where the codeword is to be stored by the use of the pair of macros 'consider', 'insert' i.e. the block must be identified for future reference.

Example: To allocate a block which is to hold an array of 20 numbers. The array is to be named by the identifier Mine

```
array      Mine;   (introduced universally)
 :
 :
begin      example;

      ALLOC [ <+21>, <+1>, <+0>, <+0> ]
      consider [ <+Mine> ]
      insert
end   example;
```

8.4.1 The SPAN macro 'consider '

An example is given above of the use of the SPAN macro 'consider'. In general, consider is used to identify a block. It is never used on its own but in conjunction with other SPAN macros which specify the operation to be performed on the block so identified.

The 'consider' always consists of the mnemonic 'consider' followed in square brackets, by a parameter which specifies the address of the location containing the codeword.

In the example, 'consider' is followed by the macro 'insert' which specifies the operation i.e. store the codeword in the location specified by the previous 'consider'.

8.5 Reference to Elements of A SPAN Block

The previous paragraphs explained the allocation and naming of a single dimensional array. To reference elements of the array the SAC programmer makes use of the SPAN macros

```
tk1 [ A,I ]
stl [ A,I ]
```

With these macros tk1 and stl are mnemonics (i.e. pre-assigned). The macro parameters A and I are identifiers; A is the array identifier and I is the location which contains an integer specifying which element is required, i.e. the index.

Examples

1. Store the numbers 10 to 29 in the 20 locations of the array allocated in the last example.

```
begin  example
data   number, element;
       30  <+10>
       20  number
       02  0
       20   element
next)  30  number
       stl  [ Mine, element ]
       22  number
       32  element
       05  <+20>
       41  next
        .
        .
        .
end     example;
```

2. Store the number 15.7 in the 1st, 4th and 7th elements of the array.

```
begin  example;
       30  <+15.7₁₀>
       stl  [ Mine,<+1> ]
       30  <+15.7₁₀>
       stl  [ Mine, <+4> ]
       30  <+15.7₁₀>
       stl  [ Mine, <+7> ]
end    example;
```

The above example illustrates two points:

a) The diamond bracket constant form can be used for identifier I.

b) The number to be stored must be brought into the accumulator each time a stl instruction is given.

3. Sum the numbers stored in the array.

```
begin  example;
data   sum;
       26  sum
       02  0
       20  I
next)  tk1 [ Mine,I ]
       24  sum
       32  I
       05  <+20>
       41  next
        .
        .
        .
end    example;
```

8.6 Two Dimensional Arrays

8.6.1 Allocation

In Chapter 4.2 it was stated that for a two dimensional array the main codeword points to a block of codewords each of which points to the elements of one row of the array.

To allocate storage for a two dimensional array the SAC programmer must adopt the following procedure.

Identify the array by use of the control name array. This allocates storage for the main codeword, and names the array for future reference.

Allocate a block to contain the codewords of the rows of the array and inform SPAN that the codeword of this block is the main codeword.

Allocate a separate block for each row of the two dimensional array and insert these codewords in the appropriate positions in the codeword block.

To insert a codeword into a codeword block use is made of the SPAN macro,

row [I]

in conjunction with the macros consider and insert.

The macro row [I] is used to specify which element of the codeword block is being referred to. I is replaced with either a diamond bracket constant or by the identifier of the location which contains the index.

```
e.g. consider [ <+Matrix> ]
     row [ <+2> ]
     insert
```

would cause a codeword to be placed in the second element of the codeword block.

Example

Allocate storage for a matrix of n rows and m columns assuming that n and m are stored in locations "N" and "M" respectively.

```
array   Mine;
data    M,N;
begin   matrix;
data    number;
        02  N                                 (set the size of the codeword block in 'number')
        20  number
        ALLOC [number,<+3>,<+0>,<+0>]         (Allocate a codeword block of length N+1)
        consider [<+Mine>]
        insert                                (name the array Mine for future reference.)
        02  0
        20  which                             (set 'which' to indicate the first row.)
        02  M
        20  col                               (set the size of each row in col)
again)  ALLOC [col,<+1>,<+0),<+0>]            (Allocate a block for each row of the matrix)
        consider [<+Mine>]
        row [which]
        insert
        32  which
        05  N
        41  again
         .
         .
         .
end     matrix;
```

8.6.2 Reference to Elements

In paragraph 8.5 it was explained that the macros

```
tk1  [ A,I ]
st1  [ A,I ]
```

were used to reference elements of a single dimensional array. Similarly for a 2-dimensional array the macros

```
tk2 [ A,I,J ]
st2 [ A,I,J ]
```

are used, where I contains the index of the row and J contains the index of the element within the row that is required.

Example

Add the 1st row to the second row of the matrix allocated in the last example.

```
                02  0
                20  J
repeat)         tk2 [ Mine, <+1>,J ]
                20  temp
                tk2 [ Mine, <+2>,J ]
                60  temp
                st2 [ Mine, <+2>,J ]
                32  J
                05  M
                41  repeat
                  .
                  .
                  .
```

8.7 Multi-dimensional Arrays

Multi-dimensional arrays are allocated by an extension of the method described in the previous paragraph.

Elements of the array are however accessed in a somewhat different manner.

Thus to access element $a_{1,5,10}$ of an array $a_{i,j,k}$ one would write

```
consider  [ <+a> ]
row       [ <+1> ]
row       [ <+5> ]
row       [ <+10> ]
take
```

A similar method is used to store elements but in addition the SAC programmer must remember to place the item in the SPAN location 'reserve' before obeying the macro consider

```
e.g.      20 reserve
          consider [ <+a> ]
          row  [ <+1> ]
          row  [ <+5> ]
          row  [ <+10> ]
          store
```

This is necessary as the macro consider destroys the contents of the accumulator.

8.8 Deletion of Arrays

Just as arrays can be allocated they may also be deleted dynamically. Whenever a SAC programmer has no more use for an array he should delete it so that the store space occupied is released. Thus store space is then available for later allocation.

To delete an array the SAC programmer writes the macro consider followed on the next line by the macro 'delete'.

```
e.g.      consider [ <+Mine> ]
          delete
```

Note that in a multi-dimensional array the above pair of macros would delete the whole array. To delete a single row of a two-dimensional array the programmer should write

```
consider [<+A>]
row [ I ]
delete.
```

8.9 Summary

In the previous paragraphs the use of the following SPAN macros has been explained.

```
ALLOC [ size, type, status, offset ]
consider [ <+A> ]
insert
tk1 [ A,I ] , st1 [ A,I ] , tk2 [ A,I,J ] , st2 [ A,I,J ]
row [ I ]
take
store
delete.
```

A full description of all the faciliti es of SPAN is given in Section 2.5.2. of the 503 Manual. Insert, take, store, and delete are examples of parameterless macros. A call of any such SPAN macro must be preceded *immediately* by a description of the SPAN unit that is being operated upon. This description consists of either a single 'consider' or else a 'consider' followed by one or more calls of 'row'.

see PG

CHAPTER 9

SUBROUTINES WITH PARAMETERS

In Chapter 7.4 it was explained that to enter a subroutine the macro

enter [A*B,n]

was used where n is the number of parameter words after the enter.

The parameters can take any whole word form described in Chapter 6.8.

9.1 Parameter Macro

Within the subroutine the parameters, which are written after the 'enter' macro, are accessed by means of the 'parameter' macro.

The 'parameter' macro consists of the mnemonic 'parameter' followed by the number of the parameter in square brackets. e.g. parameter [1] will bring the first parameter after the 'enter' macro into the accumulator.

Example

To find the sum of squares of two numbers whose addresses are given by two parameters following the enter instructions.

```
begin   square;
data    mod,x,y;
        parameter [ 1 ]
        20  mod/30 0
        20  x
        63  x
        20  x
        parameter [ 2 ]
        20  mod/30 0
        20  y
        63  y
        60  x
        exit [ 3 ]
end     square;
```

The entry to this subroutine would be in the form

```
data    length,width;
begin   master;
          .
          .
          .
        enter [ square,2 ]
        +length
        +width
          .
          .
          .
end     master;
```

On exit from the subroutine the sum of squares of the contents of length and width would be in the accumulator.

The address of an array identifier may also be used as a parameter but in this event care must be taken as the array identifier contains a *codeword* and not data.

9.2 Storage of Codewords

There is one basic rule that must always be obeyed when using SPAN, "SPAN must have access to all codewords".

This is obviously necessary to enable SPAN to update codewords when relocating blocks in store.

This rule can be restated as:-

"SPAN must know the addresses of all basic codewords".

The SAP2 compiler informs SPAN of the addresses of the program codewords and the <u>array</u> declarations enable SAP to inform SPAN of the addresses of the data basic codewords.

Thus for the programmer the rule can be expressed "Copies of codewords must not be held in data space to which SPAN has no access".

Because of this rule, and because it is often necessary to transfer information from one section of program to another, the following convention has been adopted:-

The address of the codeword, and not the codeword itself is transmitted from one section of program to another.

The macro parameters normally adhere to this convention. Thus the consider macro is written.

consider [<+A>]

the diamond brackets indicating that the address of the codeword of A is required by SPAN.

The exceptions to the *convention* are the macros tk1 [A,I], st1 [A,I], tk2 [A,I], st2 [A,I] .

They are written in this form for optimisation purposes i.e. tk1 [A,I] is the optimised version of

```
consider [ <+A> ]
row      [ I ]
take
```

(Note that this form does not break the basic *rule* as the codeword is being transmitted to SPAN itself).

9.3 Array Identifiers as Parameters of Subroutines

Array identifiers may be used as parameters of subroutines.

Thus one can write:

```
        array   A,B,C;

        begin   master;

                enter [ sum,2 ]
                +A
                +B
                +C
        end     master;
```

where 'sum' is a subroutine to store the sum of two arrays.

Within the subroutine the parameters are accessed by the parameter macro. However, the values of the parameters must be stored in locations
the relevant

Thus the subroutine would start:-

```
        begin   sum;

                parameter [ 1 ]
                20  X
                parameter [ 2 ]
                20  Y
                parameter [ 3 ]
                20  Z
```

X,Y,Z now contain the *addresses* of the array codewords.

In order to sum the matrix it is necessary to determine the length of the array by use of the consider and length macros.

```
        e.g. consider [ X ]
             length
```

This places the length of the array in the accumulator as an integer.

Note: that we write

See CC

```
        consider [ X ]
```

because X is a formal parameter of the macro and *contains the address* of a codeword.

To access a location of the matrix we write tkl [X,I].The formal introduction of X informs the SAP2 compiler that X contains the address of a codeword and not the codeword itself.

i.e. tkl [A,I] is a conditional macro dependent on the form of introduction of the identifier A.

Thus the subroutine becomes:

```
begin   sum;

data    temp, I,size
        parameter  [ 1 ]
        20  X
        parameter [ 2 ]
        20  Y
        parameter [ 3 ]
        20  Z
        consider  [ X ]
        length
        20  size
        02  0
        20  I
again)  Tk1  [ X,I ]
        20  temp
        Tk1  [ Y,I ]
        60  temp
        St1  [ Z,I ]
        32  I
        05  size
        41  again
        exit [ 4 ]

end     sum;
```

The SAC programmer should note that parameter [u] is a macro and therefore represents a set of instructions which bring the parameter into the accumulator. It is *not* an identifier which represents the parameter.

Therefore the programmer *cannot* write

consider [parameter [u]]

since this would generate a portion of code as a parameter of consider.

CHAPTER 10

UNIVERSAL IDENTIFIERS

In chapter 3.3.1. mention was made of universal identifiers. These are identifiers which have meaning outside the program itself.

An essential universal identifier is the name of the program itself which is used for external reference to the program. The program name is introduced by use of the control name program followed by the identifier terminated by a semi-colon.

```
e.g. program EDITAL;
```

This is the first introduction of any program.

The other essential universal identifiers are the list of entry points to a program. These are introduced by the control name trigger which is the last control name of any program. The entry points are listed after trigger and must have been previously introduced within the program by a block introduction. The entries in the trigger list are separated by commas and terminated by a semi-colon.

```
program   THIS;
block     alpha, beta (one);

trigger   alpha, one*beta, beta;
```

These identifiers are universal labels.

It is sometimes useful to be able to refer to data and arrays of one program from blocks of another program. This is possible if universal data and arrays are used. A universal data identifier is introduced by the control name universal. Two programs which wish to use the same universal data must use the same universal identifier.

must declare all arrays universally.
The programmer ~~may also have universal arrays and~~, these are introduced by the control name array given imediately after the program introduction. Global arrays are introduced after the block introductions.

Thus:

```
program    compute;
array      all;
universal  many;
block      alpha,beta (one);
```

In this case 'compute', 'all' and 'many' are universal identifiers. 'alpha','beta' are global identifiers.

CHAPTER 13

REPLACEMENT STATEMENTS

13.1 This chapter describes how whole sections of SAC may be replaced by a single identifier, provided that the identifier is properly defined as a "replacement identifier" and the "text" i.e. the code it is itended to represent, is specified in the manner described below.

Replacement identifiers may be local or global but not universal in scope. The standard macros are equivalent to universal replacement identifiers but have fixed meanings and the SAC programmer cannot change their universal effect (although their local meaning can be altered, see 13.4) .

Replacements are most useful when a particular section of program is to be obeyed many times. A simple example of the use of replacements is given by the sequence of instructions required to form twice the content of the accumulator plus one. The instructions are

```
55 1
04 <+1>
```

and can be introduced as a replacement text for a replacement identifier, say SOCK.

Subsequently, SOCK may be used in place of, and will be replaced by, the above instructions. Thus to perform

$$C := (2A + 1) + (2B + 1)$$

where A and B are locations containing the operands and the result is placed in a location specified by C, the SAC could be

```
30   A
SOCK
20   C
30   B
SOCK
24   C
```

13.2 Form in which replacements are specified

Specification is in two parts

(a) introduction of the identifier

and (b) specification of either the replacement text, or the name of the peripheral (e.g. the second tape reader) from which this text may be drawn whenever it is required for insertion into the program .

Both parts of the specification must be given together in an introduction statement. An introduction is given by writing the control name replace followed by as many replacement specifications as required in the order

(i) the replacement identifier
(ii) an equals sign
(iii) a letter representing the "source" of the replacement text.
(iv) if the letter was S, then the actual text enclosed in square brackets.

The complete statement is terminated by a semicolon.

SAP recognises four letters as references to sources from which a replacement text may be obtained:

S	the store
R	the second reader
T	the typewriter
Z	none.

Thus a section of code held on paper tape may be used as a replacement text for the replacement identifier SETUP after giving the introduction

```
replace     SETUP = R ;
```

providing that the tape on which the text is punched is placed in the second reader when a call of the replacement is met during assembly.

The introduction statement for the example in 13.1 could be

```
replace     SOCK = S [ ; 55  1
                         04  <+1> ];
```

When the source is S, the store, the "=S" may be excluded from the replacement statement, thus

```
replace     SOCK [ ; 55  1
                   04  <+1> ];
```

Several replacements may be specified in one introduction statement, e.g.

```
replace  SETUP = R, PRINT = R, Mod 17 = T, SOCK [ ; 55  1
                                                    04  <+17> ] ;
```

Note that the text is not inserted when the introduction statement is read but every time the replacement identifier is read in the program thereafter. In the above example, each time the replacement identifier PRINT is read during assembly (after the introduction has been made) the tape in the second reader is searched until

PRINT [. . . .*replacement text*. . . .]

is found. Other replacements on the tape are ignored. The required replacement text is then inserted into the program and SAP proceeds with assembling the next instruction.

This provides a convenient method of assembling a program from its constituent subroutines.

Whenever during assembly SAP encounters a replacement identifier which has been given the source T, a text is requested by display of a message on the typewriter. In the above example

```
TYPE Mod 17 [
```

is displayed. SAP then waits for a text to be typed in, terminated by a closed square bracket.

The operator may type the character = which means the replacemen t text is a single data or label identifier, with the same name as the replacement identifier.

Incorrectly typed texts may be cancelled by typing the non-escaping character vertical bar (V). The text then has to be retyped from the beginning.

A source of Z, for a replacement text means that whenever the corresponding replacement identifier is read no insertion is made and SAP passes on to the next instruction. This is most useful when the replacement for a particular identifier is not required in one block.

The replacement text for any replacement identifier can be changed by re-introducing it at a later stage in the program. The new text is then inserted for all subsequent calls of that replacement identifier; previous insertions are not affected.

Certain identifiers are so placed that they can only be interpreted as replacement identifiers, e.g. identifiers on lines by themselves. These are ignored by SAP if unintroduced when read.

As with data identifiers, replacement identifiers may be made local or global by introducing them inside or between blocks, but they may not be universal in scope.

13.3 Replacement texts - parameters

Any section of SAC may be used as a text; it can be a single identifier or function, or a complete section including data-, label - and even replacement-introductions. Each text must always be enclosed within square brackets and preceded by the appropriate identifier. It is important that this arrangement is adhered to even when texts are drawn from sources other than the store

e.g. if the introduction

```
      replace SOCK = R ;
```

is made (see 13.2) then the tape in the second reader should contain

```
SOCK [ ; 55 1
         04 <+1> ]
```

If the same replacement text is used serveral times in different parts of a program, it is often necessary to be able to change the names of certain data and label identifiers appearing in the text.

The text employs identifiers of four different types:

1. Label, data and block identifiers

2. Replacement-identifiers. Replacements may call each other to any depth, although a replacement cannot call upon itself, as this would lead to an infinite recursion - see paragraph 13.4.

3. Standard macro names.

4. Parameter identifiers. These may be "renamed" when the replacement text is inserted into a program. They are referred to as "formal parameters" when in the text, and the renamings and replacements as "actual parameters", using ALGOL terminology (Vol. 2.1.3). Permitted re-namings (actual parameters) are:

(i) an integer, from which the sign may be omitted.

(ii) a constant enclosed within diamond brackets, which is equivalent to the address of a location containing this constant.

(iii) another identifier. 3 part identifiers e.g A*B+C may not be used as replacement parameters (except in the standard macros entr, trans etc).

Formal parameters, if any, are enclosed in the same square brackets with their text and need no other introduction. The parameters are listed first and separated from the text by a semicolon. e.g.

```
replace CALC [ factor, step ;
               42  inc
               53  factor
        inc)   20  result
               30  step
               24  value ] ;
```

In this example, 'factor' and 'step' are the formal parameters. The separating semicolon must always appear before the replacement text even if there are no parameters.

The actual parameters are given when calling the replacement, and are listed after the replacement-identifier enclosed in square brackets, in the order set up by the formal parameter list.

Hence

```
CALC [ 5, <+2> ]
```

result in the following instructions being inserted into the program

```
        42  inc
        53  5
   inc) 20  result
        30  <+2>
        24  value
```

Care must be taken when the same replacement text is used more than once in one block, to ensure that labels introduced in the replacement text are made parameters. Also when the replacement is called, the actual parameters for the labels must be different each time the replacement is required within that block. If this is not done an error will result since effectively there are two identical labels in one block.

Thus facilities are provided for the systematic replacement of sections of SAC programming by mean of a single identifier, possibly with parameters. The effect of a sequence of such statements can be determined by systematically making the replacements. The expanded text, thus constructed, must obey the rules of syntactically correct SAC.

13.4 Identifiers which are given more than one meaning

If the same identifier has been introduced more than once under different control names, and it occurs in a position which makes its meaning syntactically ambiguous SAP determines which reference is intended according to the following rules.

1. The meaning of an identifier met at the beginning of a new line, which is neither a label nor a signed identifier, is determined in the order of priority

    a local replacement
    a global replacement
    a standard macro

    Thus, the instructions represented by a replacement identifier are inserted into a program when a call of that identifier is read, in preference to those represented by a standard macro of the same name.

    One important factor must, however, be considered before applying this rule. Once such an identifier has been used as ~~is found to be~~ a standard macro name, it becomes equivalent to a local replacement in that block and any further reference to it is taken as a reference to the standard macro.

2. The meaning of an identifier occurring in the address position of an instruction is determined in the order of priority

    a formal parameter
    a local replacement
    a global replacement
    an ordinary identifier

    An identifier in this position is never taken to be a standard macro.
    The following examples should clarify these rules.

    (i) Given the local introduction at the beginning of block B

```
replace consider [ A ;
                   24  X
                   41  Y
                   40  A ] ;
```

In the same block, B, a call of

```
        consider [ P ]
```

is interpreted as the replacement given above. The operation of the standard macro does not take place. If the same identifier was used in another block of the program, the macro would operate as usual.

(ii) A replacement's identifier can be quoted in its own text, either explicitly or else as part of the text of one of the nested replacements. Since a replacement cannot call upon itself, the meaning of such an identifier would be taken to be an ordinary identifier or a formal parameter. e.g. given the introductions

```
data     A;
replace  A [ ; 30 A ] ;
```

The instruction A

is taken as the replacement identifier. The A in the text of the replacement cannot be a formal parameter and therefore must refer to the ordinary identifier.

```
30  A
```

is inserted into the program where A now represents the data identifier.

(iii) Given the introductions,

```
data     A, B ;
replace  A [ ; B ]
```

in the instruction 30 A

A is taken to be the replacement identifier A, since this takes precedence over the data identifier A.
The instruction thus becomes

```
30  B
```

(iv) The same identifier can be used as both a formal and an actual parameter, without producing odd effects.

e.g. After the replacement introduction

```
replace  PLANT [ SEED, SOIL ;
                 30  SEED
                 24  SOIL ] ;
the instruction PLANT [ SOIL, POT ]

becomes          30  SOIL
                 24  POT
```

i.e. Although the formal parameter SOIL becomes actual parameter POT this does not affect the actual parameter SOIL.

(v) Given the introduction

```
replace whichone [ ;<+1> ], thisone [ ;<+2> ], GET [ whichone; 30 whichone ] ;
```

the instruction GET [thisone]
becomes 30 <+2>
SAP starts by assembling the text of the replacement identifier GET i.e. 30 whichone. The identifier whichone is interpreted as a formal parameter, rather than a replacement identifier, so GET [thisone] becomes 30 thisone (not 30 <+1>). Then thisone is replaced by its text <+2>.

(vi) A formal parameter only operates on identifiers actually written in its own replacement text i.e. the actual parameters of any nested replacement call will only take the actual values of the immediately outer replacement, they will not be affected by the actual parameters of other outer replacements. e.g.

Given the introduction

```
replace  SUB [ ; 53 A ], TEST [ A ; SUB
                                41  A ];
```

```
A call of    TEST [ error ]
becomes       53   A
              41   error
```

The identifier A in the text of the replacement identifier SUB, is not affected by the actual parameter, error, according to the rule given above. Thus, when determining the affect of a replacement which itself contains a replacement, the text should be expanded one layer at a time, applying the appropriate set of parameters to each layer. These parameters should also be applied to the actual parameters quoted in the call of any identifier which appears in the layer considered e.g. Given the introduction

```
replace   SUB [ B ; 53 B ],  TEST  [ A, B ;
                                   SUB [ A ]
                                   41    B ] ;
```

and a call of TEST [<+15>, error]
and expanding the replacement one layer at a time it becomes

```
          SUB [ <+15> ]
          41  error
```

and then

```
          53 <+15>
          41 error
```

This rule excludes such a case as

```
replace  HOP [ A, B ; . . . . ] ,  SKIP [ ; HOP ] ;
```

with a call SKIP [error, <+1>]

since SKIP has been given actual but not formal parameters.

CHAPTER 14

CHAPTER INTRODUCTIONS AND SAC PROGRAM FORMAT

14.1 Chapter introductions

It was stated that the blocks of a SAC program could be grouped into chapters. A chapter occupies a single SPAN block and thus if a number of SAC blocks are combined into one chapter they will be moved round the store together. If one of the blocks of the chapter is in the main store then all other blocks of that chapter are also in the main store. Thus it is usual to group SAC blocks which are used in conjunction with one another into a chapter.

This grouping is done by means of the block introductions. Thus all blocks named between the control name block and the following semi-colon constitute a chapter.

For example the introductions

```
block  alpha (one), beta (two);
block  gamma;
```

would ensure that alpha and beta constituted one chapter while gamma was in a chapter on its own.

14.2 SAC Program Format

Each SAC program is constructed of three main parts.

(1) The program begins with a 'head' giving all universal and generally all global identifier introductions, starting with the program name, followed by the universal data identifiers and then the block names and global data identifiers.

(2) The blocks come next, each being limited by the control names begin and end with global data introductions between them if necessary. Blocks may not intersect or be contained in each other and every program must be made up of at least one block. A semicolon must follow each introduction statement (including those in the 'head') and each end.
The blocks may include SAC instructions, macros of SPAN, PCP etc. and local label and data introductions.

(3) The program finishes with the control word trigger followed by a list which specifies the entry points of the program. These entry points must have been introduced as global labels or be block names.

14.3 Summary of control names

The following introductions must always be given.

program, block , trigger

Every block of program must start with the control name begin and conclude with end.

The following control names may be used to introduce universal data identifiers

universal, array

2.4.1.14.

The following control names may be used to introduce global data identifiers.

data, array

The following control names may be used to introduce local data identifiers.

data

The control name replace may be used to introduce replacement identifiers, local or global depending on where the introduction is given.

Order of giving instructions

program array universal	universal
blocks data replace	global

begin
data
replace
end
	local

trigger	universal

CHAPTER 15

COMMON PROGRAMS

The previous sections have been concerned with writing individual programs.

It has been explained that identifiers are used to name data and blocks within a program. However, the job to be programmed may be too large and a team of programmers may be working together. If the job is written as one large program the individual programmer would have to ensure that they did not use the same identifiers for different blocks or global data identifiers. Therefore facilities are provided so that the job can be written as a set of programs. In this case the individual programmers only have to ensure that they use the same names for the universal data items which they all wish to use.

The facilities provided are:

1. Universal data and array introductions
2. 'trans' and 'enter' macros to other programs.
3. Links so that complete programs can be used as subroutines.

trans and enter macros

To transfer to another program the trans and the enter macros are written in the form

```
trans [ A*B*C ]   enter [ A*B*C,n ]
```

where A is a global label or block B of program C. A*B must have been given in the trigger list of C.

The macros ntrans,ztrans,otrans can be given in the same way.

The enter macro must only be used if block B of program C is left by an exit [m] instruction.

see P.E

SYMBOLIC ASSEMBLY MARK 2

APPENDIX

This appendix provides the specification of SAP2, Issue 2 in so far as it differs from SAP1. A complete specification of the language SAP2 is obtained by reading this in conjunction with the description of SAP1 (503 Manual 2.3.1). Any reader unfamiliar with Symbolic Assembly Code would be advised to read issue 2 of the SAP2 Programming Guide (2.4.1) first.

Part 1 contains the specification of SAP2, Issue 2, Part II the operating instructions and Part III the format of an assembled program.

I. SPECIFICATION

SAP2 makes provision for programs which are too large to fit into the mainstore with all their dataspace. This has been done by incorporating SAP into the SPAN/PCP system, and the resulting changes and additions to the SAP1 specification are described in the following section. A program written in SAP1 is acceptable to SAP2 provided that it uses no reader or typewriter replacement,(*) and that the COMP and EXITCP facilities are used to reference only programs assembled by SAP1 (e.g. standard binary tapes): all other COMP instructions should be replaced by the "enter" instruction and the SAP1 program will then be acceptable to SAP2.

A. UNIVERSAL IDENTIFIERS

SAP1 programs can only communicate with each other via the COMP instruction, and this type of subroutine entry may only be made to a program which is already in store at assembly time. This facility may be used in any SAP2 program, though only to enter a program which has been assembled by SAP1; and it will be found that any SAP1 common program entered via COMP must be in binary form, as there is not sufficient room to hold SAP1 and SAP2 in store together. SAP2 programs, however, always communicate with each other via universal data, label and array names; and as before, no program may be referred to before it has been input or assembled into the store.

A local identifier may be used anywhere inside the block it is introduced in,(but not outside), while global identifiers may be used anywhere in the program they belong to. A SAP2 universal identifier may be used in any program which is read into store, and, according to its type, must be introduced in one of the ways described below.

A universal data or array name must be introduced in the head of the program which first uses it. It may subsequently be introduced in the heads of any of the other programs which use it, though this is not an essential thing to do, but only a precaution. All universal data and array names are simple(i.e. are single identifiers)and the introductions, if made, must appear in the following order:

*) Chapter 12

```
program E;
array A1, ... ;
universal U1,U2, ... ;
(block)
```

These universal data and array identifiers may be handled just like ordinary data names, though an array location may only be used to hold the codeword of some block of space obtained from SPAN at runtime. It is essential that the codeword of each SPAN-block obtained at runtime be placed in such an array location, and that the array locations are used for no otherpurpose. Note that array introductions may only be made universally.

Any global label in a program can be made universal by introducing it in the program's trigger list, and all the global labels mentioned in a program's trigger list are considered to be universal. Outside the program to which it belongs, a universal label may only be used as a parameter of one of the special macros 'enter', 'trans', 'chapter', which are detailed below. In this case, the reference must have one of the forms

B*B*P, L*B*P

where L stands for a label name,
B stands for a blockname,
P stands for a programname.

A three-part identifier of the above form may not be used anywhere as a parameter in a programmer's own replacement statement.

B. CHAPTERS

SAP1 assembles each program into 2 blocks, one of which contains the program instructions, and the other the program dataspace; there is no way of shifting and relocating such a program in the store after it has been assembled. By using the facilities of SPAN, SAP2 is able to assemble the program instructions into a number of SPAN storage blocks, in such a form that the instructions may be relocated at runtime by SPAN, if this should be necessary (to free space in mainstore). A set of SAC blocks may be stored in one SPAN storage block called a "chapter", and the "chapter-structure" of any program may be defined in its block introduction list by grouping together in separate introductions the names of the blocks which are to form a single chapter.

Thus, the introductions

```
block A,B(L1,L2);
block C(L2);
```

at the head of a program indicate that the program has 3 blocks, which are to be stored as 2 independent chapters: of these, one is to contain blocks A and B, and the other block C. The program blocks must be arranged in chapters on the source (mnemonic) tape, and the program chapters must appear in the same order as their associated block introductions; for example, given the above introductions, the blocks could be punched in the order A,B,C or B,A,C but not as C,A,B or A,C,B etc.

Only the current chapter need be in the mainstore at runtime; the others, being independent and relocatable, may be in main or backing store, or on a magnetic tape. There are 2 ways of bringing a chapter into mainstore:- by transferring control to it, as described below, or by giving an explicit instruction to SPAN. The latter method cannot make use of the SPAN macro-instructions

```
consider  [A]
fast
```

because there is no way of naming a chapter, but the same effect may be achieved with the macro-instruction "chapter [A]". This is really the same macro as consider[A], and has the same kind of effect, except that A is not the name of the SPAN block being considered, but the name of one of the global or universal labels in the chapter. Thus, the instructions

```
chapter[SPIDER*SPIDER]
fast
```

will bring the chapter containing block SPIDER down into the main store (if it is not already there). In this example, the block SPIDER is in the same program as the calling chapter; to bring down the block SPIDER in another program WEB a 3-part identifier as required:

```
chapter[SPIDER*SPIDER*WEB]
fast
```

Note that when the program containing the above instruction is being assembled, the program WEB must already have been assembled.

chapter[A] is appropriate wherever consider[A] is appropriate, e.g.

```
chapter [R]
fix
```

will fix the chapter containing block R in the mainstore.

C. TRANSFER orders and SUBROUTINES

A complete relocation system would ensure the adjustment of every reference to each program word which is moved from one point to another in the store. The system described here, however, is only able to cope with adjustments to those references which lie in the same SPAN chapter as the point referred to,since a full relocation system would be unacceptably slow. Therefore, although the instruction

```
40A
```

may be used when transferring control from one point to another in the same chapter, it may not be used when transferring control from one chapter to another, since A is a runtime variable. Inter-chapter jumps are effected through special macros which are able to call on the system to calculate the current position of any label to which a transfer is to be made.

The simple transfers to an instruction labelled A (and A must be a global or universal label) arewritten in SAP2:

```
 trans[A]     (≡40A,44A)  unconditional
ntrans[A]     (≡41A,45A)  negative
ztrans[A]     (≡42A,46A)  zero
otrans[A]     (≡43A,47A)  overflow
```

The trans macros may also be used to transfer control to points in the current chapter, and SAP2 will assemble such a call in one of two ways. If the label A and the trans[A] instruction lie in the same chapter, then a simple jump of the form 40A will be assembled. If the trans[A] instruction and the label A lie in different chapters, then a small set of instructions (a macro) will be assembled, which will cause control to be passed at runtime to a special routine in SPAN, in which the current position of A will be calculated; and if necessary, the chapter containing the label A will be called into mainstore before the transfer is made. Because of this optimisation of trans instructions, it is recommended that all transfers between SAC blocks be written in this form, so as to allow a greater flexibility to the program's chapter structure.

The SAP1 instructions

SUBR, A and EXIT, m

may be used when entering at or exiting from a point in the same chapter, and

COMP, A and EXITCP, m

when entering and exiting from a program assembled by SAP1; such programs must be in a binary form.

Entry to any chapter in any program in main or backing store may be effected with the SAP2 macros

enter[A,N] and ENTER[A]

A, here, is either a global or a universal label, and N specifies the number of parameter wholewords, if any, which have been written under the enter instruction (to specify e.g. a print format). The enter macro is assembled as 3 wholewords, and the accumulator, overflow and auxiliary registers are preserved when it is obeyed. The ENTER macro is assembled as 2 wholewords, and is obeyed in about one half the time of the enter macro, though only the accumulator register will be preserved.

Once in the destination chapter, the nth of the parameter wholewords written by the programmer under an enter macro may be brought to the accumulator with the macro

parameter[n], $1 \leq n \leq N$. (current versions of SPAN impose the restriction $N \leq 8$)

This parameter macro can only access the N parameter words stored under the last enter to be activated.

The programmer should arrange for each "enter" or "ENTER" instruction to be paired with an

```
exit[m]
```

instruction, just as the SUBR instruction is paired with EXIT,m and the COMP instruction with EXITCP,m. (Note, though, that "exit" is not a recognised abbreviation for "exit[1]".)The enter and ENTER macro cause link information to be stored in the LINK location of the block containing label A. This makes it possible for the "exit" macro to be paired with a SUBR instruction, so that the same subroutine may be triggered with an "enter", an "ENTER" or a SUBR instruction. However, EXIT may not be paired with enter or ENTER, nor is any variation allowed in the pairing of COMP and EXITCP.

The 'enter' and 'ENTER' instructions may be used to make subroutine entries to points in the same chapter. Such an 'enter' instruction will never be optimised to a SUBR, in the way that 'trans' is sometimes optimised to a simple jump. This allows entry to be made to a subroutine in the same chapter in which some action is taken which results in the chapter being shifted; for example, control might be passed from the subroutine to another chapter, in which space is booked. The ordinary SUBR-EXIT combination does not cater for such a chapter shift, while the enter-exit combination does.

The restriction explained at the beginning of this section should be carefully noted: no machine code instruction may refer directly to a label in another chapter. For example, the instruction 30 <+B> is erroneous if B is a block in some other chapter. Also an improved facility: there is no important restriction on the use of compound (i.e. identifier with added integer) addresses; the identifier part may be unallocated at the time the compound reference is met, even when used inside diamond brackets.

It is important that the parameters of the macros

```
enter[A,n]
trans[A]
n/z/ortrans[A]
chapter[A]
```

should not refer to a program which is unavailable to SAP2 when the calling program is being assembled.

D. INTRODUCTIONS

The introductions in a SAP2 program must be made in the following order:

<u>program</u>
<u>array</u>
<u>universal</u>
<u>block</u>

<u>block</u>
<u>replace</u>
<u>data</u>
<u>replace</u>
<u>begin</u>
<u>replace</u>
<u>data</u>
<u>replace</u>
<u>end</u>
<u>trigger</u>

These instructions are optional, except for <u>program</u> <u>trigger</u> <u>block</u> <u>begin</u> <u>end</u>, which must be made once each. Note also that global items may only be introduced at the head of a chapter; they may not be introduced between the blocks of a chapter.

E. RESTRICTIONS ON REPLACEMENTS MACRO INSTRUCTIONS ALLOWED

The following restriction applies to all programs assembled by SAP1 and SAP2; an identifier which has been introduced as a replacement may not subsequently appear between any underlined basic name and the terminating semicolon. Also SAP2 does not recognise reader or typewriter replacements.

The following lists detail all the macro instructions permitted in a SAP2 program as standard replacements i.e. they may be quoted without the formality of an introduction in a replacement statement.

1. PCP and SPAN Macros

SAP2 functions within the PCP/SPAN non-basic system, i.e. at least 1 unit of core backing store must be included in the configuration. A full description of the meaning and use of the macros listed below can be found in 2.5.2 and 2.5.3 of the 503 Manual, the only exceptions being the SPAN macros Tkl[a,b] and Stl[a,b] which have been included recently:

Tkl[a,b] and Stl[a,b] may be used like the macros tkl and stl except that the actual value of parameter a is to specify the name of a location which contains the address of the SPAN codeword of the array referred to by the macro. In tkl[a,b] and stl[a,b] the parameter a specifies the name of some location which contains a codeword. For example, the two instructions:

```
        tkl[array,count]
and     Tkl[<+array>,count]
have the same effect.

SPAN:=  alloc[A]                      PCP:=   open device[D,N,EL]
        ALLOC[a,b,c,d,]                       close device[D,N,EL]
        consider[a]                           shut device[D,N,EL]
        row[a]                                cancel device[D,N,EL]
        take                                  buffer[D,N,B,F,EL,M,C,K]
        store                                 BSUPPLY[<+n>]
        tkl[a,b]                              BRETURN[BUFFER]
        tk2[a,b,c,]                           return
        stl[a,b]                              restart[label*block]
        st2[a,b,c]                            stop
        Tkl[a,b]
        Stl[a,b]
        insert
        detach
        delete
        length
        lowbound
        newbounds[a,b]
        slow
        fast

SPAN:=  setfile[a,b]
        cwsetfile[a,b]
        file[a]
        cwfile[a]
        popup[a]
        cwpopup[a]
        chopfile[a]
        cwchopfile[a]
        fixed
        free
        read only
        write also
        retire
        recall
        banish H
        collapse
        setorigin[H,K]
```

2. SAP2 Special Macros

```
ENTER[A]
enter[A,n]
exit[m]
parameter[K] (K ≤ 8 in current version  of SPAN)
trans A
n/z/otrans[A]
chapter[A]
```

where A is a global or universal label.

3. SAP1 macros Allowed in a SAP2 Program

```
SUBR,A
EXIT
EXIT,m
COMP,A,n                (A must have been assembled by SAP1)
EXITCP
EXITCP,m
STOP
N/Z/OSTOP
RAPprint
RAP1print
RAPread
RAPsearch
RAPiprint               (which displays a 4 digit integer from
                        the accumulator)
```

II. OPERATING INSTRUCTIONS

(This describes the operating procedure for SAP programs which use SPAN but not PCP; a more general method is outlined in sections 3.2.1 and 3.2.2 of the overall description.) The procedure is as follows: the steps must be executed in the order given:

(i) CLEAR THE STORE

INPUT RAP

SET PROTECTION

(ii) INPUT SPAN BY TYPING IN.

The rule is invariable; SPAN must always be stored from location 5 on-wards.

(iii) INPUT SAP2 by TYPING IN.

SAP2 occupies about 2900 locations, and, throughout its use, stays fixed in the position to which it has been input.

INPUT ANY PROGRAMS WHICH HAVE BEEN CODED IN RAP OR SAP1 BINARY

For example PTSREAD or EDIT8. Such programs may not be input once step (iv) has been made.

(iv) TYPE SPAN;b. (where b units of CBS available).

This effectively divides the store into 3 SPAN-blocks each with a SPAN-codeword generated for it in SPAN's workspace. The 3 blocks are: 1) the low addressed store containing SPAN, SAP2 and some of the binary programs input during step iii, (2) the freestore area, and 3) the high addressed store containing RAP and the rest of the binary programs input during step iii. It also sets the appropriate headerword in core-backing store.

(v) LOAD A SAP2 PROGRAM IN READER 1

SET THE WORD GENERATOR

TYPE SAP2.n.m. (other possible entries are detailed below)

The general format of a stored SAP2 program is: a main SPAN-codeword block (containing the RAP head), a fixed SPAN-block for workspace and constants, the program chapters (which are relocatable), a relocatable trigger chapter, and some universal dictionary information. Bits 39, 38 and 34 on the word generator are used by SAP2 for the same purposes as SAP1.

Bits 37,36 and 35 have no effect on the assembled program.

A program's relocatable chapters are always assembled directly onto backing-store, so that SAP2 issue 2 cannot be used on machines without backing-store.

SAP2 must be given an estimate of the amount of data-space used in any program it is required to assemble, and of the number of distinct diamond bracket constants to be expected there. These are the numbers "n" and "m" typed after the trigger to SAP2. Although these are not easy numbers to estimate, it will usually be found sufficient if some fairly generous approximation is given. SAP2 will end each assembly with the message

```
END    <FF>  <LF>
EST    <n>   <m>
```

in which the displayed values of n and m are the smallest acceptable.

More exactly:

n = the amount of global and local dataspace introduced in the program, plus the number of blocks, plus the number of chapters, plus 2.

m = the number of blocks, plus the number of global labels, plus the number of diamond bracket constants which do not refer to a label or a blockname.

If one of the typed estimates is found to be too small, one of the 2 messages "X data" and "X cnst" will be displayed, according to whether n or m is the faulty number. SAP2 will then re-use the reserved space to check the rest of the program (though the message "X cnst" is almost sure to be followed by a number of spurious error indications) and the corresponding estimate printed out at the end of the run will be too small by the number of times the corresponding error message has been displayed, times the amount of the original estimate.

Programs for which m and n are accurately known may have these estimates punched into their object tapes, thus: <u>program</u> NAME(N,M);
The appropriate entry to SAP2 is then

SAP2.

If it is now required to override the punched tape with a typed estimate the entry is

SAP2;1.n.m.

Programs may also be edited as they are assembled. To do this, type an edit tape in the EDIT8 style, make sure that EDIT8 is input at stage iii, load the edit tape in reader 2, and type either

```
               SAP2;2;n;m.
        or     SAP2;2.
```

(according to whether the estimates are to be read from tape or typewriter).

Note that although the displayed freestore limits indicate a genuine freestore area, the "first-free" location always contains the size of this area, and should not be altered by any program other than SAP2 or SPAN.

TRANSLATE IN THE SAME WAY ALL OTHER SAP2 PROGRAMS REQUIRED FOR THIS RUN

(vi) TRIGGER PROGRAMS AS IN THE SAP1 SYSTEM

The procedure for triggering programs which use PCP is given in section 2 of the note on PCP.

The CANCEL;X. message is replaced by SAP2;3.X. when using SAP2.

III. FORMAT OF AN ASSEMBLED PROGRAM

The second issue of SAP2 gives its object programs the following store-format, (the program is stored as a SPAN-array):-

(i) BLOCK A, always 15 locations long, which contains the main SPAN-codewords for the assembled program-array.

i.e.

4 codewords, for block A

block B (for universal data and array space)

block C (for the chapter codewords program dataspace and block LINKs "label-pseudo-codewords" - see iii 2 - and constants)

block D (for the trigger chapter)

+0 (not used)

700:000 (which is a "breaker", to tell SPAN that there are no more codewords in the higher addressed locations of the block).

a RAP program head:

(5 locations) (Note that the program is not given a RAP store-sumcheck; the first location of the head is therefore made to contain a "label-pseudo-codeword" for the beginning of the program's trigger block).

a LINKCP location

& 2 locations to contain a trans instruction to the trigger chapter.

(ii) BLOCK B, for universal array and dataspace.
i.e. a location for each universal array item which has been newly introduced. (These are arranged in alphabetical order.)

a "breaker" (700:000 - see i)

a location for each universal data item which has been newly introduced. (These are arranged in alphabetical order.)

(iii) BLOCK C, for chapter codewords, data and constant space.
This block is divided into two parts:

1. The first part contains n locations (where "n" is the first of the 2 integers typed after the name SAP2.) In these locations are placed:-

   a codeword for each chapter, (of the form 40 257+r:00 x, where r is the number of "relocation words" planted before the chapter, and x is the address of the last of these relocation words, (see iv).

   a "breaker" (700:000 - see i).

   a location for each global data item introduced before the beginning of the first program block, arranged in order of introduction.

   a LINK location for each block of the program; these LINKs happen to be arranged in alphabetical order: the lower case alphabet is considered to lie alphabetically <u>after</u> the upper case alphabet.

   a location for each global or local data item introduced in or after the first program block, arranged in order of introduction.

   If too large a value was given for "n", there will follow a number of spare locations.

2. The second part contains m locations (where "m" is the second of the 2 integers typed after the name SAP2). In these locations are placed:-

a "label-pseudo-codeword" for each blockname and each global label read in the block instruction. These label pseudo-codewords are used by the enter,ENTER and trans macros at runtime, and they have the form 00 y:0 $\genfrac{}{}{0pt}{}{0}{4}$ Z where y is the address of the chapter codeword of the label, and z is the relative position of the label within the chapter - the first word of program in the chapter has relative address 1.

those "diamond bracket constants" used in the program which do not refer to label or blocknames.

If too large a value was given for "m", there will follow a number of spare locations.

(iv) The program chapters. (These will be on backing store after assembly) This is the format of a chapter:

a headerword (see SPAN 2.5.2)
a special word, for use by SPAN
r relocation words:- one relocation word is planted for every 19 words of program.
the program proper.

(v) BLOCK D This is a relocatable trigger block, which is entered whenever the program is triggered. It has the form of a chapter.

(vi) BLOCK E This contains information about the universal items introduced in or on behalf of the program. Note that block E contains its own codeword.

Blocks A B and C will be packed at the high addressed end of store.

BLOCK E is at the low addressed end.

The program chapters will be found on backing store after assembly, but are brought into store as the program is run.

SECTION 3: 503 FORTRAN IV

CONTENTS LIST

Chapter 1

1.1 Introduction

The FORTRAN system implemented on the 503 conforms to the ECMA standard with certain extensions and relaxations. This document does not set out to describe the language in detail. It is intended to be used in conjunction with a copy of the draft ECMA standard on FORTRAN which will be supplied on request.

The FORTRAN system is intended for use on a 503 installation which has one or more units of core backing store and magnetic tapes.

1.2 503 FORTRAN representation

Programs may be punched on cards or on paper tape. In both cases the full character set as specified by the ECMA standard (section 3.1) is acceptable. In addition the currency symbol £ may be used.

Programs read from paper tape should be punched on an 8-channel flexowriter according to the 503 Paper Tape Code conventions.

Programs read from cards should be punched according to the conventions of the 503 Elliott Card Reader Code. (See Volume 1, Part 4, Section 6 of the 503 Technical Manual for details.)

Chapter 2

2.1 Language Description

The FORTRAN IV compiler is implemented according to the rules of the ECMA standard with the following exceptions.

2.1.1 The EQUIVALENCE statement has no effect.

2.1.2 All subprograms must be translated before the first sentence to them is translated.

2.1.3 A dummy array argument can only be replaced by an array name. Any occurrence of an array element name in an argument list is treated as an expression. Also the dummy array and the actual array must:

(a) have the same number of dimensions

and (b) each dimension has the same limit in the two arrays.

2.1.4 The main program must start with a comment line. The first six characters will be used as the program name. These characters must all be alphanumeric, and the first one must be a letter.

2.1.5 Array subscripts may be any integer expression. There is one exception to this. Evaluation of an array element in an input/output list must not cause a transfer to a subprogram so in this case the subscript expressions cannot include external function references.

2.1.6 The following additional input/output statements are accepted:

Statement	Interpretation
READ f,list	Read from Paper Tape Reader 1
PRINT f,list	Output on line printer
PUNCH f,list	Output on Paper Tape Punch 1

2.1.7 The Statement IF (SENSESWITCH j)L_1, L_2 is accepted. L_1 and L_2 must be statement labels and j is an integer constant such that $1 \leq j \leq 19$. The effect of the statement will be that control will be transferred to L_1 or L_2 depending on whether key j of the F2N2 keys of the word generator is depressed or not.

2.1.8 In the statements 'PAUSE n' and 'STOP n' n is treated as a string of from one to five characters. When the statement is obeyed the characters are displayed on the output typewriter.

The PAUSE statement causes the running program to wait. The program may be continued by changing the sign key on the word generator.

2.1.9 Any number of continuation lines (cards) are allowed.

2.1.10 Arithmetic expressions (see 6.1 page 10 of Draft ECMA standard on FORTRAN).

The standard rules here are relaxed. It is possible to have mixed code arithmetic.

(i) By use of any arithmetic operator except / two operands of type integer may be combined to form a result of type integer

(ii) By use of any arithmetic operator except ** two operands of different types (or both real) may be combined to form a result of type real. (Note (integer) ** (real) is not defined).

Warning: This means that the result of I = $^4/_3 + ^5/_4$ is 3 and not 2.

If immediate truncation after an integer division is required the non-standard intrinsic function "DIV" should be used.

2.1.11 For input/out statements the ECMA standard applies with two exceptions:

(i) For output with the FORMAT setting E w.d the output always has the decimal point appearing after the first digit.

(ii) Each device has a standard physical record length associated with it. Logical records (i.e. records specified by the user) are artificially segmented into physical records of standard length. The appearance of a / in a FORMAT results in one physical record being skipped.

2.1.12 The standard list of intrinsic functions is increased by one. This is the function DIV (a_1, a_2) where a_1 and a_2 are both of type integer and the result is $a_1 \div a_2$, also type integer (see note in 2.10).

2.1.13 The standard list of basic external functions is increased by one. This is the function TAN(a) where a is of type real. The result if of type real.

Note: for all trigonometric basic external functions the argument is assumed to be expressed in radians.

2.1.14 The control variable of a DO loop must not be a COMMON element.

2.1.15 GOTO or ASSIGN statements may not contain COMMON elements.

Chapter 3

3.1 Limitations

3.1.1 Array subscripts must be $\leq$ 4094.

3.1.2 Any common block (including unlabelled common) must be $\leq$ 4094.

3.1.3 The total number of program units must be $\leq$ 17.

3.1.4 Integers must be in the range

- 274 877 906 944 to + 274 877 906 943

Real numbers must be in the range

$- 5.79 \times 10^{76}$ to $+ 5.79 \times 10^{76}$ approx.

3.1.5 All assignment statements must have = sign punched on the first line of the statement.

3.1.6 Subprograms must not have more than 63 dummy arguments.

3.1.7 No executable program may have more than 32 named COMMON blocks.

3.1.8 The number of consecutive slashes in a FORMAT statement must be $\leq$ 31.

3.1.9 The number of digits in the fractional part "d" of an E or F FORMAT descriptor must be $\leq$ 15.

3.1.10 The total field width specified in a FORMAT descriptor must be ≤ 255.

3.1.11 The repeat count preceding a descriptor in a FORMAT s tatement must be ≤ 511.

3.1.12 The modulus of a scale factor specified in a FORMAT statement must be ≤ 127.

Chapter 4 Input/Output Statements

4.1 The standard FORTRAN input/output statements may be used directly by the programmer for transfer of information. The devices which may be used and the modes of transfer allowed are listed in Table B.

Each device has a standard record length associated with it. If the input/output statement demands the transfer of a larger record then as many standard records as are necessary to exhaust the list are transferred.

Warning

Care should be exercised in skipping records which are larger than standard size. The slash (/) in a FORMAT statement results in one physical record being transferred and if a logical record consisting of n standard length records has to be skipped then n slashes will be required. The same warning applies for the BACKSPACE statement.

Table A contains the full list of standard FORTRAN input/output statements plus three others which the 503 FORTRAN system will accept.

TABLE A

Statement	Effect
1. READ (n,f) list	Read from FORTRAN device number n in character mode. If the list is empty then one physical record (≤ standard record size) is skipped.
2. READ (n) list	Read from FORTRAN device number n in odd parity binary mode. If the list is empty one physical reoord is skipped.
3. WRITE (n,f) list	Write on FORTRAN device number n in character mode. If the list is empty then one empty record is transferred.
4. WRITE (n) list	Write on FORTRAN device number n in odd parity binary mode. If the list is empty then one empty record is transferred.
5. READ f, list	The same as 1 except that n is assumed to indicate paper tape reader 1 i.e. n = 1
6. PRINT f, list	The same as 3 with n assumed = 5, i.e. line printer.
7. PUNCH f, list	The same as 3 with n assumed = 3, i.e. paper tape punch 1

TABLE B

Data can be read from, or written to, the following devices in binary or character form, according to the following table.

FORTRAN DEVICE NUMBER	DEVICE	Read Binary	Read Character	Write Binary	Write Character	Standard record size
1	Paper Tape Reader 1	Yes	Yes	No	No	120
2	Paper Tape Reader 2	Yes	Yes	No	No	120
3	Paper Tape Punch 1	No	No	Yes	Yes	120
4	Paper Tape Punch 2	No	No	Yes	Yes	120
5	Lineprinter	No	No	No	Yes	121
6	Elliott Card Reader	No	Yes	No	No	80
7-12	Magnetic Tape Handlers 3-8	Yes	Yes	Yes	Yes	501

The standard record length is in terms of 503 words. The packing density for character mode is normally one 7-bit character per word. For magnetic tape it is 5 7-bit characters per word.

The input/output is done via the Peripheral Control Program (PCP).

Standard length records are allocated for all the devices (see Table B). If the FORMAT control specifies a record of greater length than the standard size then the record is artificially segmented. On output any record of standard size which has not been filled will be cut down so that no spurious blank characters are output. On input any record of less than standard size is accepted. If, however, a record of greater than standard length is read all but the first n words will be discarded. (n = standard length record).

4.2 Special Effects

1. On reading from paper tape a newline L character will be interpreted as an end of record indication and no further characters will be read.

2. Only 500 of the 501 words of the standard magnetic tape record are used for transfer. The first word is used by PCP for administration purposes.

3. Only 120 of the 121 words of the standard lineprinter record are transferred. The first word holds the character used to control vertical format. This has to be set by the FORTRAN source program for each logical record specified by the FORMAT statement.

   The characters are as follows:

Character	Effect
blank (space)	Throw one line
0	Throw two lines
1	Output on New page (top of form)
+	Output on same line

If the FORMAT statement specifies a logical record longer than 120 characters the first 120 characters are output and the 121st character is placed in the first word of the next physical record so determining the vertical format for this record. If this is meaningless according to the above table, one line will be thrown.

4. Repeated slashes in the FORMAT statement have the following effect:
   (i) On input the rest of the current record is ignored and the following n-1 physical records are by-passed.
   (ii) On output the current record is output and n-1 empty records are output.

5. If data written on magnetic tape is later required to be read then a REWIND instruction must be given for that handler. A manual rewind is not sufficient.

6. ON input leading spaces are ignored, and subsequent spaces are treated as zeros.

7. Spaces, slashes and Hollerith strings in output FORMAT statements are not output unless they are followed by an output element.

4.3 Input/Output error messages

The following error conditions are reported at run time.

4.3.1 If an illegal attempt (as specified in Table B) is made to transfer to any device then the message DEVICE ERROR is output on the typewriter and control returns to RAP. It is not possible to continue.

4.3.2 All the error conditions recognised by PCP are reported by the FORTRAN system. The message displayed is CLASS X ERROR Y DEV NO Z

where X = class of error as specified on page 30 section 2.5.3 of the Technical Manual
Y = type of error as specified on same page,
and Z = device number

For all errors except class 1 no continuation is possible. By changing the sign key it will be possible to continue after a class 1 error.

4.3.3 If a record >501 words is discovered on magnetic tape, the message LONG RECORD n is displayed where n is the handler number.

It will be possible to continue by changing the sign key but some information will be lost.

4.3.4 The End of Tape condition is detected and the message END OF REEL n displayed where n = handler number. It will be possible to continue by changing the sign.

4.3.5 The messages

(i) PRINT ERROR and

(ii) IPRINT ERROR

(iii) RPRINT ERROR

(iv) TYPE ERROR

are displayed in the following circumstances:

(i) Incorrect FORMAT specification on trying to print a floating point number in E or F FORMAT. The program comes to a keyboard wait and, on changing sign, the number will be printed in FORMAT E, 15.9 and the program will continue.

(ii) Incorrect FORMAT specification on trying to print an integer in I FORMAT. The program cc es to a keyboard wait and, on changing sign, the number will be printed according to FORMAT I 13 and the program will continue.

(iii) An attempt has been made to print in E or F FORMAT a real element which is not in standard floating point form. No continuation is possible.

(iv) FORMAT descriptor does not agree with type of element to be input/output.

4.3.6 The message DATERR n may be displayed on trying to read a number. No continuation is possible.

In the following explanation

d = number of digits after decimal point as specified by FORMAT descriptor.

w = total number of characters as specified by FORMAT descriptor.

n = 1 (a) $d \geq w$.

(b) Number of digits after decimal point $\neq$ d

(c) decimal point or E in integer.

(d) two decimal points in number.

(e) decimal point after E in number.

(f) two E's in number.

(g) no decimal point or E in real number.

	(h)	null integer strings before and after decimal point.
	(i)	no number before E
n = 2	(a)	illegal number character input
	(b)	two ± signs on number
	(c)	± sign in middle of number
	(d)	± sign last character according to format
n = 3	(a)	integer overflow
	(b)	real number overflow
n = 4	Wrong character found on trying to read a logical element.	

4.4 REWIND, BACKSPACE, ENDFILE

These instructions all apply to magnetic tape only. Any attempt to apply them to any other device will result in DEVICE ERROR being displayed at run time.

The effects of the instructions are as follows:

(a) REWIND n where n is an integer constant or integer variable ($7 \leq n \leq 12$)

(b) BACKSPACE n
This instruction causes the handler identified by n to retreat one <u>physical</u> record.

(c) ENDFILE n
A tape mark is output on the handler identified by n. Any subsequent attempt to read this tape mark results in an error message CLASS 4 ERR1 being displayed. It will be possible to continue by changing the sign digit.

Chapter 5 Error Indications

5.1 Compile Time Error Messages

The following error conditions in the source program are detected and reported by the FORTRAN system.

The message ERROR X STAT Y is output on a new line on the lineprinter.

X = Error number

Y = Statement number

The contents of an input buffer are also output on a new line on the lineprinter.

If the lineprinter is in the manual state when an error message is to be output then

LP IN MANUAL

will be output on the typewriter, and continuation of error output takes place automatically when the lineprinter is taken out of the manual state.

If there is no lineprinter attached to a machine, or if the line printer is switched off, then the error message will be output on the typewriter.

In all error conditions except Error No. 75 (program too large) the rest of the source code is checked for further errors.

In addition to the above mnemonic errors the messages SPAN ERROR 1 or SPAN ERROR 2 may be displayed. This means that the Storage Planning and Allocation Scheme is unable to find space in main store or backing store respectively. No continuation is possible.

ERROR NO.	DESCRIPTION OF ERROR
01	No left hand side in assignment statements.
02	Error in IF statement. No arithmetic or Logical operator found.
03	Error in READ or REAL statement. Cannot determine type of statement.
04	Error in WRITE statement. Cannot determine type of statement.
05	Statement not allowed in 503 FORTRAN.
06	Incorrect character in label.
07	Character not allowed in FORTRAN.
08	Line contains more than 72 characters.
09	Parenthesis not matching.
10	Array with more than 3 dimensions.
11	Empty Parenthesis.
12	Identifier name does not begin with letter.
13	Character in column 7 should be a letter.
14	Label on continuation line.
15	Identifier declared twice.
16	Two incompatible definitions of identifier.
17	Error in subscript of array declarator.
18	Left parenthesis not followed by identifier in statement function definition.
19	Identifier not followed by comma or right parenthesis in statement function definition.
20	Right parenthesis not followed by = in statement function definition.
21	Comma not followed by identifier in statement function definition.
22	More than 63 parameters listed in statement function definition.
28	In COMMON statement, identifier appearing twice in a COMMON block.
30	Class inconsistent in array declarator.
31	Dimension inconsistent in array declarator.

ERROR NO.	DESCRIPTION OF ERROR
32	First limit inconsistent in array declarator.
33	Second limit inconsistent in array declarator.
34	Third limit inconsistent in array declarator.
36	Array element has subscript which is too great.
37	Error in separator between subscripts in array declarator.
38	Subscript limit > 4094 in array declarator.
41	Non α/n character other than $\underline{L}$ ( ) , / = + * . **
42	Identifier other than END followed by £
43	Full stop followed by non α/n character.
44	Character other than full stop following full stop and an identifier.
45	Unidentified operator between two full stops.
46	α character other than E following digit(s) of a number.
47	α character following E following digit(s) of a number.
48	Non α/n character other than ± following E following number.
49	Another E following a real number already containing E.
50	Integer constant too large.
51	Decimal exponent of more than 2 digits.
52	Same label used twice.
53	A non FORMAT label has been used previously as a FORMAT label in a READ or WRITE statement.
54	Jump not allowed to this label from outside the DO loop (Forward Jump).
55	This statement causes jump into closed DO loop (Backward jump).
56	The statement (READ,WRITE,...) refers to a label which is not a FORMAT label.
57	In GOTO statements and GOTO assignment statement, label is not a digit string.
59	Unidentified item after the label in an Unconditional GOTO.

ERROR NO.	DESCRIPTION OF ERROR
60	In Assigned GOTO i, (K1, ... KN) and Computed GOTO (K1, ... KN),i i should be an identifier.
61	Comma missing after identifier in an Assigned GOTO.
62	Parenthesis missing in ASSIGNED or COMPUTED GOTO.
63	Comma missing before identifier in Computed GOTO.
64	In Assigned and Computed GOTO, and in GOTO assignment statement, i should be a simple integer variable reference.
65	In ASSIGN k to i, i is not an identifier.
66	Error in ASSIGN k to i statement.
67	Symbol following target label in DO statement is not an identifier.
68	Control variable of DO statement not an integer variable.
69	No = sign in DO statement.
70	No comma after parameter in DO statement.
71	Parameter of DO statement is not an integer constant or identifier.
72	Last statement of DO loop is RETURN,PAUSE,STOP,DO,GOTO or arithmetic IF.
73	Dimension of LABELLED COMMON block > 4094.
74	Dimension of UNLABELLED COMMON block > 4094.
75	Program too large to compile.
76	NO (after FORMAT.
77	Incorrect symbol following separator in FORMAT specification.
78	Unknown conversion code in FORMAT specification.
79	, following) or , in FORMAT SPECIFICATION.
80	Integer followed by incorrect symbol in FORMAT specificatinn.
81	Incorrect symbol following one or more slashes in FORMAT specification.
82	, followed by one or more slashes in FORMAT specification.
83	Non-integer following E,F,I or L in FORMAT specification.

ERROR NO.	DESCRIPTION OF ERROR
84	Incorrect specification following E or F in FORMAT specification.
85	Field descriptor followed by incorrect symbol in FORMAT specification.
86	Field width, repeat count or number of consecutive slashes outside permissible range in FORMAT specification.
87	Incorrect character in subprogram definition.
88	Subprogram definition not followed by identifier.
89	Subprogram name not followed by (or $\underline{L}$.
90	Function with no parameters.
91	Formal parameter not an identifier.
92	Symbol after formal parameter not) or ,
93	More than 63 formal parameters in a subprogram definition.
94	Inadmissible continuation line in Hollerith string.
95	In EXTERNAL statement, item not an external function nor a subroutine.
96	In EXTERNAL statement, item not in dictionary.
100	Incorrect character(s) in label in arithmetic IF statement.
101	Symbol other than , or $\underline{L}$ following label in arithmetic IF statement.
102	No integer following SENSESWITCH in IF (SENSESWITCH....... statement
103	SENSESWITCH integer j outside range $1 \leq j \leq 19$
104	More than 32 different COMMON block names used.
105	Logical IF statement followed by DO statement, Logical IF statement or any non-executable statement.
110	First statement of program is not a COMMENT or subprogram definition.
111	END statement is not preceded by GOTO, STOP, RETURN or Arithmetic IF statement.
112	Statement in program body is not END, FORMAT or executable statement.
113	No identifier following CALL.
114	CALL OF intrinsic function.
115	Reference to unintroduced subprogram.

ERROR NO.	DESCRIPTION OF ERROR
116	A dummy argument which is not of class "subprogram" has been used in a subprogram reference.
117	CALL of function or function reference to subroutine.
118	No actual arguments in subprogram reference.
119	No (following instrinsic function reference in actual argument list.
120	Actual argument of class array or subprogram not followed by (,) or , .
121	Actual argument of impermissible class.
122	Recursive call of subprogram.
123	Actual argument which replaces dummy array argument has not been declared.
124	Actual argument which replaces dummy array argument is not an array.
125	Actual array argument is not of same type as dummy array argument.
126	Actual argument which replaces dummy subprogram argument has not been declared.
127	Dummy argument which has been defined as being of class external subprogram has not been replaced by an actual parameter of correct class.
128	Dummy argument of class external function has not been replaced by an argument of the correct class.
129	Dummy argument of class external function has been replaced by an actual function of incorrect type or incorrect number of arguments.
130	Dummy argument of class subroutine has been replaced by an argument of incorrect class.
131	Actual argument in subprogram reference is not followed by , or).
132	Insufficient arguments in subprogram reference.
133	Too many arguments in subprogram reference.
134	Symbol of incorrect class at start of actual argument in a subprogram reference.
135	Second symbol of expression in actual argument is incorrect.

ERROR NO.	DESCRIPTION OF ERROR
136	Type of simple variable used as actual argument is not the same as the dummy argument type.
137	Subprogram name is same as previously defined subprogram.
138	NO (after READ or WRITE statement.
139	Device parameter is not a +ve integer constant or an identifier.
140	Device parameter integer constant is not in the range 1-12.
141	Device parameter identifier is not of integer type.
142	Incorrect separator, i.e. not , or) in READ/WRITE statement.
143	Not) , (or $\underline{L}$ after ARRAY identifier in the list.
144	Not) , or $\underline{L}$ after a list.
145	Not enough) , in list.
146	Too many)s in list.
147	No) after DO - implied loop in list.
148	Incorrect character in PRINT or PUNCH statement.
149	Undeclared label.
150	COMMON statement terminated by a single /.
151	Incorrect symbol following / in COMMON statement.
152	Incorrect symbol in COMMON statement.
153	Incorrect separator between two elements of a COMMON statement (not / or ,).
154	COMMON block name not followed by /.
155	Sixth character of apparent COMMON statement is not an N.
156-199	Unallocated.
200	In sub expression the first symbol is not: left parenthesis, subprogram name array name or dummy argument.
201	Matching parenthesis of a sub expression not found.
202	No = sign found.
203	External function name on left hand side of assignment statement is not name of current program unit.
204	Right hand side of assignment statement terminated by symbol of incorrect class.
206	Empty arithmetic expression.
207	Logical operand in arithmetic expression.
208	Expression starts with operator other than + or -

ERROR NO.	DESCRIPTION OF ERROR
209	Element of incorrect type in arithmetic expression.
210	Identifier other than simple variable, external or statement function, array or dummy argument in left hand side of an assignment statement.
211	Missing operand.
212	No left parenthesis in array element.
213	No right parenthesis in array reference.
214	Number of subscripts used in array reference different from number declared.
215	Result of a subscript expression not of type integer.
216	Empty Logical expression.
217	Non-Logical operand in Logical expression.
218	Logical Sub-expression does not start with: Arithmetic identifier, Logical identifier, constant, arithmetic operator, .NOT or left parenthesis.
220	Inadmissible operand in Logical expression.
221	Impermissible start to Logical expression.
222	Right parenthesis missing in Logical assignment statement.
223	No Relational operator in Relational expression.

5.2 Run Time Error Messages

The following run time error messages are always detected:

(1) Assigned GOTO statement has not had one of listed labels assigned.

(2) Computed GOTO statement has had a value assigned to its control variable which is out of range. For both these errors the message

RUN ERROR 1 is displayed.

(3) Recursive call of a subprogram results in the message RUN ERROR 2 being displayed.

(4) INTOFLO - integer overflow
log error - attempt to take log of number ≤ 0
exp error - attempt to evaluate e^a where $a > 254 \log_e 2$
trig error - parameter of SIN,COS or TAN is $> 2^{28}$
sqrt error - try to take square root of -ve number.
It will be possible to continue after these messages by changing the sign digit.

The following error conditions are detected by the Storage Planning and Allocation Scheme:

(1) No room in main store for program and/or data block.

SPAN ERROR 1 is displayed.

(2) No room in backing store for program and/or data block

SPAN ERROR 2 is displayed

It is not possible to continue in either case.

On a checked run SPAN also detects the following error state:

Subscript value in array reference is out of range.

This gives rise to one of the messages

```
SPAN ERROR 4
SPAN ERROR 5
```

Again, no continuation is possible.

Chapter 6 Format for punching source program

The source program may be punched on either paper tape or cards.

6.1 Cards

The format of the source program is as specified in the ECMA standard. The last 8 columns will be ignored. There is no limit on the number of continuation cards.

6.2 Paper Tape

The format of the source program is exactly the same as on cards with the following exceptions:

(i) The character space is used instead of blank.

(ii) The character $\textcircled{\underline{L}}$ is used to indicate the end of a line. If more than 72 characters are punched on any one line effect is undefined.

(iii) The character tab $\textcircled{\underline{T}}$ is normally taken to mean one space. If, however, it is used in any position up to and including the sixth character of a line it is taken to indicate the end of the statement label / continuation mark field. On paper tape both spaces and tabs are taken as significant. The last line on paper tape must be followed by $\textcircled{\underline{L}}\textcircled{\underline{T}}$, not just $\textcircled{\underline{L}}$.

(iv) Blanks and erases are ignored. They do not increase the character count on a line.

(v) The character halt code $\textcircled{\underline{H}}$ halts the input tape and WAIT is displayed on the typewriter. Translation continues on a change of the sign key. $\textcircled{\underline{H}}$ does not increase the character count.

It is recommended that the standard program sheet (see Appendix A) be used.

Chapter 7

7.1 Operating System

The FORTRAN system consists of three separate programs which will normally be kept as a batch on magnetic tape. The batch will normally be called FORTRAN and will be input by BRING and LOAD. The programs of the FORTRAN systems are

(i) The Compiler (FORTRAN)

(ii) The Dynamic Routines (FORTDR)

and (iii) The Peripheral Control Program (PCP).

The system may be operated in two ways:

I Normal

II Under STAR control (see STAR description section 9.3(b)).

I. Normal mode of operation

When the system has been loaded the compiler is entered by typing FORTRAN;n. on the typewriter with the keyboard clear. The various settings of n provide the following variations

n = 1, source program is punched on paper tape and is loaded in reader 1

2, source program is read in from the Elliott card reader n = 3, source program is on magnetic tape handler 2 packed five 7-bit characters to a word and written in standard length PCP records (120 words + 1 word for block number). The mode of writing is odd parity character (PCP mode 3). The magnetic tape is assumed to be positioned just prior to the first record of the program.

4, same as 1 but subscripts will be checked at run time.

5, same as 2 but subscripts will be checked at run time.

6, same as 3 but subscripts will be checked at run time.

The entries FORTRAN;3. and FORTRAN;6. will normally be used by the STAR system but may be used normally provided the source program is written on magnetic tape according to the conventions described above.

The message RAP X at the start of compilation means RAP is corrupt.

The store should be cleared and RAP reinput.

When the END of the final program unit of the runnable program has been read then the compiler deletes itself and translation of further FORTRAN program is impossible without the re-introduction of the entire FORTRAN system.

The program name is then displayed on the output typewriter and control returns to RAP. The translated program may be entered by typing on the input typewriter, the message

PCP. program name.

Re-entry to the program can be accomplished by retyping the same message.

Normal Operation

	Action	Result
Stage 1 (a)	Ensure that batch reel containing FORTRAN system is loaded on handler 1. Put BRING & LOAD tape in reader 1 and type IN.	Part of BRING and LOAD tape input and BRING is displayed on typewriter.
(b)	Type BRING. FORTRAN.	Rest of BRING and LOAD tape is input and LOAD is displayed.
Stage 2	Type FORTRAN;n.	FORTRAN source program is translated and the program nameis displayed.
Stage 3	Type PCP. program name.	Translated program is run.

If any mnemonic errors occur during stage 2 it will not be possible to continue to stage 3.

For details of errors which may be displayed during stages 2 and 3 see, respectively, sections 5.1 and 5.2.

If the source program is being read from cards or magnetic tape the message ERRINT4 will be displayed if the particular device is unavailable. Translation continues on typing CONT;ERRINT.

If the message MISREAD is displayed then either

(i) a parity error has been discovered on magnetic tape, or

(ii) a card has been misread.

In both cases the run must be repeated from stage 1(a).

II. Operation using Segment Tape Administrative Routines (STAR)

If the translation and running of a FORTRAN program occurs as part of a STAR run then stages 1, 2 and 3 occur without any intervention on the part of the operator. There are two warnings.

(i) The running program should always be terminated by a STOP instruction.

(ii) If the running program fails to come to a STOP instruction then the STAR system will have to be reloaded.
This is accomplished by typing IN. (See STAR description, section 9.6).

7.2 Dumping of a compiled or part-compiled FORTRAN program on magnetic tape

(i) Part compiled program

If at the end of a subprogram the statement END£ is used instead of END,

```
DUMP WAIT
```

will be displayed on the typewriter. Translation can be continued on a change of sign key. However, if translation is to be interrupted and continued at a later date, the part compiled program and compiler may be dumped on magnetic tape. This is done by pressing the message button, instead of changing the sign key, and then by typing

```
FORTRAN; 7.
```

which will display

```
I
END
```

on the typewriter. The program DUMP may then be input by typing

```
IN.
```

and the store dumped on handler 1. (See description of DUMP, BRING and LOAD). If, however, NO ROOM is displayed instead of END it will not be possible to dump the store.

To continue translation the batch is brought back into store using BRING and LOAD. Then, having loaded the remainder of the program in the reader, type

```
FORTRAN; 7.
```

which will display

```
R
```

and translation continues.

503 FORTRAN CODING FORM

AUTHOR	PROGRAM	PAGE OF	DATE

1 2 3 4 5 6 7 17 27 37 47 57 67 73 80

(ii) Compiled program

At the end of translation the program name (PRNAME, say) is displayed on the typewriter. To run the program directly type

```
PCP. PRNAME.
```

However to dump the compiled program on magnetic tape type instead

```
PRNAME; 2.
```

which will display

```
I
END
```

and the store can be dumped as above.

To run the program subsequently, the batch is brought back into store using BRING and LOAD. Then type

```
PCP. PRNAME; 2.
```

which will display

```
R
```

and the program will be entered.

Operational Techniques

1. General

The philosophy of operation revolves around the use of magnetic tape as the means of storage for the relevant part of the system (see BATCH S, 2.3.2.25). This means that any awkward sequence of operations required to set up the contents of main store and core-backing store need only be performed once before that store configuration is copied to magnetic tape for fast retrieval subsequently.

The same applies to any store configuration created by making use of the systems programs, for example the users program, once translated to main store and core backing store, may be written to magnetic tape in order to avoid the necessity to repeat the assembly stage on subsequent occasions.

This section is concerned primarily with an explanation of the organisation of the component parts of the system within the store.

Certain of the component programs have been coded in a binary form by SAP Mark 1. The reasons for this vary from program to program but are either because the program concerned will always remain in main store, and thus are better left out of the SPAN store administration scheme, or because, in the case of SAP2, no advantage is gained by making full use of SPAN.

The presence of two types of coded program tapes causes no extra difficulties for the operator except in the initial preparation of the system (see part 2). All manipulation of and interchange of such programs once the system is being used takes place automatically.

The appendix to the SPAN description (2.5.2) shows that two versions of SPAN are required, the full version being automatically brought to main store (from core backing store) when found necessary. Thus the binary coded programs SETCBS and SPANSWAP have been produced and must be used when the system is initially produced, and written to magnetic tape. SETCBS enables the user to place a copy of the full version of SPAN in core backing store, and SPANSWAP, as its name implies, brings down that version in place of the restricted version when necessary. All trace of its presence in core backing store is deleted at the time of the exchange.

It should be noted that the use of BATCH S is not mandatory. Should the user develop an alternative system, provided the relevant sections of store are written away this may equally well be used.

2. Loading the basic systems tapes

The steps described below are those to be taken in preparing the basic systems batch on magnetic tape. All further batches may be built up from the stage reached below.

1. Type RESET.
2. Input SPAN (full) by typing IN.
3. Input SETCBS by typing IN.
4. Type SPAN;5.
5. Type SETCBS.N. , where N is the number of units (16,000 words per unit) of core backing store fitted. SPAN is now stored in the core backing store.
6. Type RESET.

7. Input span (basic) by typing IN.

8. Input SAP2 by typing IN.

9. Input SPANSWAP by typing IN.

10. (optional) Input other binary programs required at run-time.

11. Type span;N. where N has the <u>same value</u> as in step 5.

At this stage a batch may be dumped on magnetic tape.

Each remaining system program is coded in SAP2 form and may be assembled to store using the basic batch created above. Thus, once the basic batch has been retrieved from the magnetic tape (using BRING and LOAD), a further systems batch may be created by compiling the required program(s).

3. <u>Assembling and Running Procedure</u>

The technique for entry to the program assembled will vary according to whether or not that program uses PCP. The technique for assembly (or compilation), however, is not affected by the presence or absence or PCP. The individual variations in the method of assembly for each assembler/compiler will be found described in the appropriate section of the manual.

If the program is not to use PCP then it may be triggered in the same way as programs input under the Mark 1 system. If the program uses PCP then it must be triggered via PCP. The exact detail of the technique for running one program or two programs together will be found in the appendix to the PCP description (2.5.3).

503 TECHNICAL MANUAL

VOLUME 2: PROGRAMMING INFORMATION

PART 5: PROGRAMMING AND OPERATING AIDS FOR THE NON-BASIC 503

SECTION 2: THE STORAGE PLANNING AND ALLOCATION SYSTEM

CONTENTS LIST

1. GENERAL INTRODUCTION

The Storage Planning and Allocation system (SPAN) is designed to help the programmer make simple and efficient use of all forms of internal storage. The system allows fully dynamic allocation and reallocation of storage in blocks and permits the available storage to be used and re-used in the most economical fashion,, according to the changing needs of the program.

1.1 Dynamic Allocation of Storage Space

The programmer can use SPAN to allocate storage space dynamically so that he only uses sufficient storage for the needs of that run of the program. When the space is no longer required it can be released, for re-use by the system (e.g. buffer space in PCP). Thus, by providing a means of requisitioning storage space during the run of a program and then handing it back for re-use, much larger tasks can be effected than would otherwise be possible.

1.2 Implementation

Under the control of SPAN, the core storage of the machine is divided into blocks of consecutive words. Additional blocks may be held on magnetic tape. The first word of each SPAN block (called the *header word*) describes the block, giving its length and the type of information held in it (e.g. data, program, free space). Each block has also a *codeword* which is effectively a pointer to the position of the block in whichever form of storage it has been placed.

1.3 Program Segmentation

SPAN administers programs which are too large to fit into the main store by placing blocks of the program on core backing store and/or magnetic tape. In the case of a SAC program, SAP Mk. 2 allocates a SPAN block to each chapter of the program at compile time. The block contains not only the object code but also relocation markers which record which addresses have to be altered when the chapter is moved. Programs run under this scheme need have only those blocks which are actually being used at any given time occupying space in main storage, the rest are held on auxiliary store. When a block of the program held on auxiliary store is required it is automatically brought into main store and if necessary, some of the unused blocks moved out.

1.4 Multidimensional Structures

All data, allocated under the SPAN system, is held in blocks. Arrays are stored by means of a tree structure. For a single dimensional array there is a single codeword pointing to a data block which holds the array elements. For a dimensional array the main codeword points to a block of codewords, each of which points to the elements of one row of the array. As a result an array element in core store can be accessed by 'chaining' through the codewords, a system which is much faster than the alternative multiplicative process. A further advantage of this method of representation is that the rows need not be held adjacent to each other in the store and they can be of any length.

1.5 Single Level Storage Simulation

The operations which can be performed in main store can also be applied to blocks in backing store or on magnetic tape, and the SPAN macros used by the programmer are identical in all three cases. The SPAN system does this by keeping in fixed workspace sufficient codewords to be able to determine at any moment, the position and type of storage in which every block is held. When a request is made for more space than is available in the main store, blocks are moved to backing store or magnetic tape to make room. If a reference is made to a block on magnetic tape the block is moved back into the main store. Operations for taking and storing single elements of a block in core backing are implemented by single word transfers, whereas entry to a program held on backing store initiates a block transfer to the main store. Any such movement of program and data made necessary by immediate program requirements is effected automatically.

Thus to the user, only one tape of reference is needed to access data wherever it is held, giving single level storage simulation. SPAN can be used (e.g. in ALGOL) such that the programmer need never realise he is dealing with more than one level of storage.

1.6 Storage Planning

There are facilities for more direct control allowing the programmer to specify in advance the type of storage required for his program and data. At any time during the run of the program, the most convenient place to hold an allocated SPAN block can be indicated by the programmer, giving improved efficiency of operation.

1.7 Time - sharing

Programs which consist almost exclusively of input and/or output, and contain hardly any calculation to fill the gaps, are called background programs. Since it would be uneconomic to run such programs by themselves means are provided for running several background programs at the same time as the main, foreground, program.

The normal users program is a foreground program and uses the full facilities of PCP and SPAN. When the foreground program is held up waiting for a peripheral transfer, the central processor is used by the background programs to initiate further data transfers.

In this way one foreground program can be time - shared with several background programs, but background programs must occupy a fixed area of store and must not use directly the macros of SPAN.

1.8 How SPAN is used

The programmer may write SPAN macros into any SAC program if required and the program will be translated by SAP Mk. 2 in the normal way on assembly. The SAP Mk. 2 compiler uses SPAN automatically to manage the storage of object programs.

The facilities of SPAN are also included in **ALGOL Mk. 2** and in **FORTRAN IV** but will not normally be apparent to the user, being used automatically by the translators.

1.9 Acknowledgement

Many of the methods used by SPAN are based on those described by J. K. Iliffe and J. G. Jodeit in their article 'Dynamic Storage Allocation'. (Computer Journal, Vol. 5, No. 3, pp 200-209, Oct. 62).

2. CODEWORDS AND HEADER WORDS

The Storage Planning and Allocation system is implemented by use of *codewords* which act as indirect addresses of blocks of information held in consecutive positions in some storage medium.

There is only one codeword for each block and it is always held in a block which is specially marked as containing codewords.

Every SPAN block whether it contains information or is free, has as its first location a *header word* indicating among other things the length of the block and its contents i.e. real numbers, program, codewords or integers, etc., or whether the block is free.

For certain purposes it is more convenient that the address part of the codeword should not always point directly to the header word of the block but to some other location depending on the subscript bounds required for the elements of the block. This number is held in the codeword, together with other information; and is called the offset and has the effect that the elements of the block to which the codeword points are no longer numbered from 1 upwards but from (1 – offset) upwards.

3. ALLOCATION OF SPAN BLOCKS

Storage space is allocated dynamically in a SAC program by use of the control word array and the SPAN allocation macros. The blocks may be of any suitable length up to a maximum of 8191 locations, depending on the requirements of the user's program. The identifier which the programmer intends to use to name the block should be introduced in an array declaration. A SPAN allocation macro should then be used, which finds a block of consecutive locations of the required length and plants the header word in its first location. It then constructs a codeword appropriate to the block and places it in a special location which is called the codeword accumulator (or CWACC).

A further SPAN macro (insert) will then place the codeword in the location specified by the identifier.

3.1 ALLOC

The allocation macro is

ALLOC [size, type, position, offset]

where the necessary specification of the block is given by the four parameters.

This macro causes entry to a search routine to find a block of the required length, in the type of storage specified. When this is found, it constructs the header word and places it in the first location of the block and if the block is to contain codewords, all the elements of the block are set to zero, to prevent any previous contents being interpreted as true codewords.

A codeword for the block is then constructed to point to the, possibly non-existent, element with which the subscript value of zero would be associated, and this is placed in the codeword accumulator.

The parameters are either identifiers naming the locations containing the information or constants enclosed in diamond brackets and are given in the following order:

size	the total number of locations in the block i.e. one location for each data item and a further location for the header word.
type	this parameter can take one of four values which specify what the contents of the block are to be, with the following meanings:

1 real numbers
2 program
3 codewords
0 others (e.g. integers)

If the programmer wishes the index of each element in a SPAN block to be checked when referred to, to see that it is within the range of the block, then the integers 5, 6, 7 and 4 should be used in place of 1, 2, 3 and 0 respectively.

position this parameter also can take one of four values 0, 1, 2 and 3 which specify the type of storage in which the block should be placed initially. The following meanings are attached to the values:

1	main store
2	core backing store
3	this value is used for blocks which must be fixed in main store; for example, certain codeword blocks must not be moved, also blocks must be fixed while input/output transfers take place.
0	to be allocated in either main or core backing store, and its position thereafter being under the control of SPAN.

The position parameter can be regarded as a storage status marker.

offset the use of the offset has no other purpose or effect than to cause a shift in the index range of the block so that the elements are no longer numbered from 1 to (l-1) where l is the length of the block, but rather from (1-offset) to (l-offset-1). The offset and lower subscript bound are connected by

$$\text{offset} + \text{lowerbound} = 1$$

e.g. if lowest subscript required = – 5
then the offset = 6

This parameter must lie in the range

$$-256 \text{ to} + (8191 - 256)$$

The codeword pointer is adjusted to point to the possibly non-existent element with which the subscript value of zero is associated.

Note that codeword blocks must be allocated in main store so that SPAN can locate all codewords and update them as necessary when blocks are moved around in the store.

Examples:

(a) To request a SPAN block which is to contain 20 words of data which are to be indexed from 1 to 20. The block can be in either main store or backing store.

```
ALLOC [< + 21>, < + 1>, < + 0>, < + 0>]
```

(b) To request a SPAN block in which the number of words of data is to be as specified by the content of location 'size' and whose elements are to be indexed from 0 upwards. The block should be in core backing store.

```
02 size
20 number
ALLOC [number, <+ 1>, <+ 2>, <+ 1>]
```

4. HANDLING SINGLE BLOCKS

Each of the SPAN macros described in this paragraph, with the exception of 'tk 1' and 'st 1', must be preceded by the general introductory macro 'consider' each time one is used. This is written as

```
consider [<+ A>]
```

and indicates that a subsequent macro refers to the block or array whose codeword is held in the location with symbolic name A.

4.1 Storage of Codeword – the 'insert' macro

The action of the allocation macro is to place the codeword for the new block into the codeword accumulator, CWACC. The user's program should then transfer this codeword to another location. This enables later reference to the block to be made by reference to this location in which the codeword has been placed. The location so used must be introduced as an array identifier in a SAC program.

The macro 'insert' is provided in order to transfer the codeword from CWACC to the user's symbolic location. The macro is always preceded by the consider macro: the action of the sequence

```
consider [<+ A>]
insert
```

is to place the content of CWACC into the location specified by the identifier, A, which will necessarily have been introduced as an array identifier. Any block previously referred to by the codeword in location A is properly deleted and CWACC is set to zero. The block whose codeword has been stored as above, can subsequently be referred to by name as A.

For example, to allocate a SPAN block which is to hold an array of 20 numbers, and may be in either main or core backing store; and to name the array by the identifier Mine.

```
array   Mine;
  .
  .
  .
  .
  .
  .
  .
begin   example;
        ALLOC [<+ 21>, <+ 1>, <+ 0>, <+ 0>]
        consider [<+ Mine>]
        insert
end;
```

4.2 Deletion of Blocks

Whenever a programmer no longer requires a SPAN block he should delete it so that the store space occupied is released for further use. A block can be deleted and returned to the scheme's control by writing

```
consider [< + A>]
delete
```

Deletion of a block sets the codeword of the block to zero and all digits of the header word are cleared except those which indicate the length of the block.

Thus if the user attempts to reference a block which has been deleted by giving the address of the codeword, the SPAN routines will be unable to locate the block since there is no pointer in the location which previously held the codeword. Attempting to delete a block which has already been deleted has no effect.

When a search for space is made the header word of the deleted block indicates that the block is free and it may be re-used for other information.

4.3 Reference to Elements of a SPAN block

Each reference to an element of a block should be made by means of SPAN macros.

Suppose a block is given the symbolic name A and the index i of a particular element is held as an integer in a location which has symbolic name I: then, the following macro should be used to fetch the i th element of block A into the accumulator.

```
tk 1 [A, 1]
```

Similarly, to store the contents of the accumulator in the i th position of block A, the macro

```
st 1 [A, 1]
```

should be used.

These macros can be used to refer to blocks held in any form of storage. If the block is known to be in main store the address of an element can be computed by forming the sum of the codeword and the index of the desired element and using this in a B-lined instruction. However, there are cases in which the block referred to is not immediately accessible and the above method is not valid. (In all such cases the sign bit is present in the codeword).

4.4 Example

The Symbolic Assembly Code program below illustrates the use of SPAN macros. It reads in a set of real numbers, stores them in a data block, forms their sum and then deletes the block. The first number on the data tape is an integer showing how many real numbers are to be read and summed.

```
                                                   Comments
program    Example 1;
block      READ (integer, real), X;
array      RENOS;                         this declaration reserves the named loca-
                                          tion RENOS for the codeword of the
                                          data block – see description of SAP
                                          Mk. 2.
```

```
                                                        Comments
begin     X;
data      ITEMS, COUNT, SIZE, TYPE,
          POSITION, OFFSET, SUM;
start)    SUBR, integer * READ                reads first number on data tape as an
                                              integer.

          20 ITEMS
          02 ITEMS
          20 SIZE                             SIZE holds the length of block required.

          02 0
          20 TYPE                             = 1 i.e. real numbers.
          26 POSITION                         = 0 i.e. initial storage left to SPAN
          26 OFFSET                           = 0 indexing starts at 1.
          ALLOC [SIZE, TYPE, POSITION, OFFSET]
          consider [< + RENOS>]
          insert                              this places the codeword of the block of
                                              real numbers thus allocated in RENOS.

          26 COUNT
read)     SUBR, real * READ                   read next real number from data tape.
          22 COUNT                            index of element read.
          st 1 [RENOS, COUNT]                 store number in required element of
                                              block RENOS.

          30 COUNT
          05 ITEMS
          41 read                             if more numbers on data tape jump to
                                              'read' next number otherwise continue.

          26 SUM
          26 COUNT
add)      22 COUNT
          tk 1 [RENOS, COUNT]                 bring to accumulator next element of
                                              RENOS.

          60 SUM
          20 SUM                              form sum.
          30 COUNT
          05 ITEMS
          41 add                              if more numbers to be summed jump to
                                              'add' otherwise continue.

          consider [< + RENOS>]
          delete                              delete SPAN block named RENOS.
end       X;
begin     READ;
          .
          .
          .
          (block to read data)
          .
          .
          .
end       READ;
trigger   X;
```

4.5 The Macro 'length'

The length of a SPAN block may be obtained by use of the macro 'length'.

```
consider [< + A>]
length
```

placed in the accumulator, as an integer, the number of usable locations in the block A (not including the header word).

This macro is most useful to the programmer, when writing a subroutine to operate on the elements of SPAN blocks, and who needs to know the length of each block he has to process.

It should be noted that the formal facility is not included in SAP2. The following macros are provided in order to provide the equivalent facility:

```
        Tk1 [X,1]
and     St1 [ X,1]
```

These will now reference the i th position of block A where I contains the value i and X contains the address of the codeword of block A.

```
i.e.  Tk1 [< + X>, I] is
equivalent to tk1 [X,I]
```

There are, however, no equivalent indirect standard macros for tk2 and st2.

The programmer may, however, include his own replacements for Tk2 and St2 as follows:

```
replace  Tk2  [A,I,J; 67 A
               30 0 :  04 I
               73 15:  41 33
               20 18/  30 0
               04 J :  41 35
               20 18/  30 0],
         St2  [A,I,J; 20 18 : 67 A
                      30  0 : 04 I
                      73 15 : 41 37
                      20 19 / 30 0
                      04 J  : 41 39
                      10 18 / 20 0]
```

5. ARRAY STRUCTURE OF SPAN BLOCKS

The natural representation of a single dimensional array is a block of consecutive locations in the store. The basic SPAN blocks are therefore convenient for this purpose. However, a multi-dimensional array would be very difficult to handle if it was necessary to deal with one row at a time each with a separate name. For this reason a special scheme of blocks is built up for representation of these arrays.

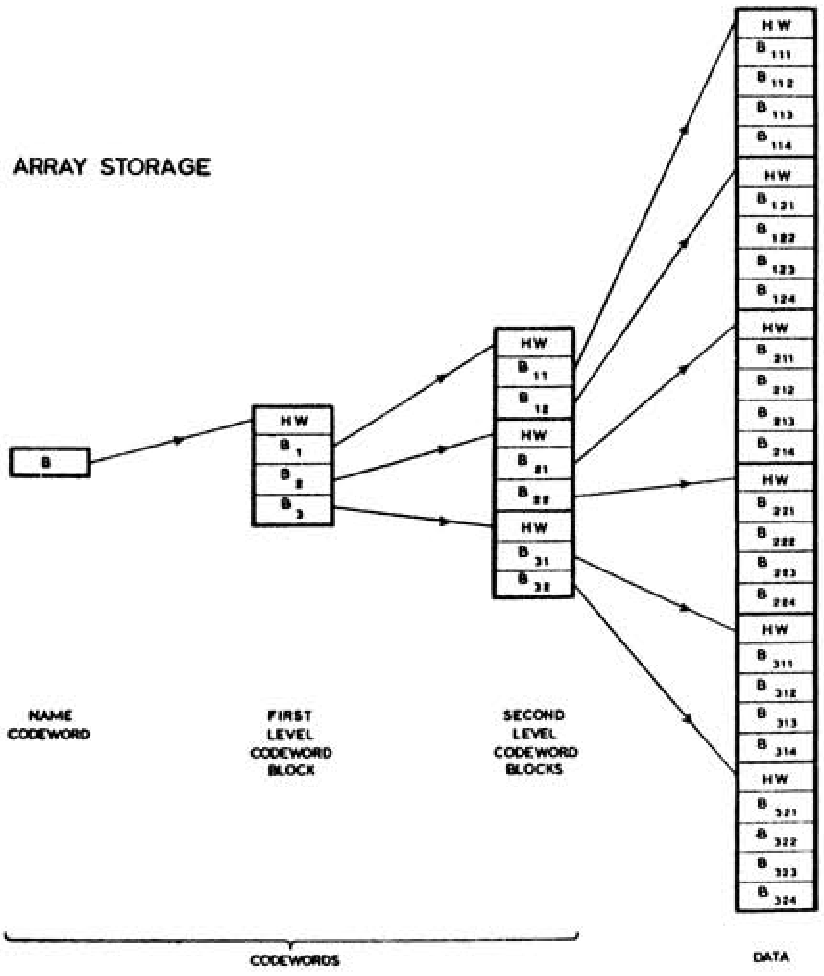
The rows are held in separate blocks which need not be adjacent to each other in the store, and may each have a different length. Thus non-rectangular arrays are stored without waste of space.

One name codeword is given to the whole array, which points to a block of codewords each of which can point to a block of codewords or data. For each extra dimension of the array another level of codewords is added between the name codeword and the data.

The macro 'row' moves one level in the codewords of the array each time it is called and corresponds to row selection.

A diagram showing the way in which a simple three-dimensional array is held in the store is given below. The array B of size 3 × 2 × 4 is shown. (For readers familiar with ALGOL terminology the array would be declared in ALGOL as:

<u>real array</u> B [1 : 3,1 : 2,1 : 4] ;)



B is a codeword pointing to the block of 3 codewords B_1, B_2, B_3. Each of these point to a separate block of codewords which in turn point to the data blocks.

Note that the codeword points to the possibly non-existent element, with which the subscript value of zero is associated.

It should also be noted that a codeword structure may not be nested to a depth of more than six.

6. ALLOCATION OF ARRAYS

To set up an array structure by program it is necessary to work from the name codeword towards the data blocks, allocating blocks on the way and placing the resultant codewords in the next higher level of codeword blocks.

6.1 To allocate storage for a two-dimensional array the SAC programmer must adopt the following procedure.

The identifier which is to be used to identify the array must be introduced, in a SAC program, by the control name array. This reserves storage for the main codeword. Then a block must be allocated to contain the codewords of the rows of the array and the name codeword for this block must be inserted in the reserved location. A block for each row of the two-dimensional array must now be allocated and the codewords inserted in the appropriate positions in the codeword block.

The SPAN macro 'row' must be used in conjunction with the macros 'consider' and 'insert' to insert the codewords into the correct positions in the codeword block.

```
e.g.    consider  [< + Matrix >]
        row       [< + 2>]
        insert
```

in which the parameter < + 2 > of the 'row' macro specifies the element of the codeword block required. The parameter could be an identifier of a location which contains the index. In the above example the contents of the codeword accumulator would be placed in the second element of the codeword block for the array 'Matrix'.

6.2 This is an example of a section of program to perform the setting up and allocation of a three-dimensional array A of size 8 × 10 × 13, in which the lower subscripts are 1, 21 and −6 respectively.

(For readers familiar with ALGOL terminology, the array would be declared in ALGOL as real array A [1 : 8, 21 : 30, −6 : 6];)

```
                                                    Comments
array   A;
        .
        .
        .
        ALLOC [< + 9>, < + 3>, < + 0>, < + 0>]      This allocates a block of 9
        consider [< + A>]                           locations which is to contain
        insert                                      codewords and places the
                                                    codeword for the block in A.
                                                    This is the name codeword
                                                    for the whole array.
```

```
                                                          Comments
        30<-7>
        20 COUNT 1
        02 0
        20 INDEX 1
REP 1)  ALLOC [<+11>, <+3>, <+0>, <-20>]                  This allocates 8 blocks of 11
        consider [<+A>]                                   locations each to contain
        row [INDEX 1]                                     codewords, the codeword for
        insert                                            each block being stored in
                                                          A INDEX 1. (INDEX 1 taking
                                                          values from 1 to 8).
        30<-9>
        20 COUNT 2
        04<-12>
        21 INDEX 2
REP 2)  ALLOC [<+14>, <+1>, <+0>, <+7>]                   This allocates 10 × 8 blocks
        consider [<+A>]                                   of 14 locations each to contain
        row [INDEX 1]                                     real numbers. The codeword
        row [INDEX 2]                                     for each block is stored in
        insert                                            A INDEX 1, INDEX 2.
                                                          (INDEX 2 takes values from
                                                          21 to 30.)
        22 INDEX 2
        32 COUNT 2
        41 REP 2
        22 INDEX 1
        32 COUNT 1
        41 REP 1
        .
        .
        .
        .
```

7. ARRAY HANDLING MACROS

7.1 Similar macros to those described for handling the elements of blocks are provided in order to take and store elements of two-dimensional arrays. These are:

tk 2 [A, I, J]

which brings to the accumulator, the element A_{ij} of the array A, and

st 2 [A, I, J]

which stores the contents of the accumulator in the j th element of the i th row of the two-dimensional array A.

In both cases I and J are the locations containing the integer subscripts i and j respectively.

7.2 In order to handle arrays of more than two dimensions, the subscripts of the element required must be specified by the macro 'consider' followed by a list of three or more 'row' macros.

The 'row' macro moves one level in the codewords of the array each time it is called and corresponds to row selection. It is written immediately after the 'consider' macro, and specifies the index number of the row in question. Use of this method of specification enables SPAN to operate on individual rows or on elements of particular rows e.g.

```
consider  [< + P >]
row       [< + 1 >]
row       [< + 2 >]
row       [< + 5 >]
```

This selects the element P_{125} of the three-dimensional array P. If P was a four-dimensional array, this would be the codeword for the row P_{125}.

These macros together with a 'take' or 'store' macro can be used to take or store the elements of any multi-dimensional array. 'take' brings to the accumulator the specified array element and 'store' plants, in the selected array element, the contents of a special location in SPAN named 'reserve' takes the absolute address 18, so that when the programmer uses 'reserve' he should include the appropriate replacement declaration in his program.

e.g. The macros below give equivalent effects but the simpler macros on the left should always be used in place of those on the right wherever possible, since they perform the same operation in very much less time.

```
tk 1 [A, I]          consider [< + A >]
                     row [I]
                     take

tk 2 [A, I, J]       consider [< + A >]
                     row [I]
                     row [J]
                     take

st 1 [A, I]          20 reserve
                     consider [< + A >]
                     row [I]
                     store

st 2 [A, I, J]       20 reserve
                     consider [< + A >]
                     row [I]
                     row [J]
                     store
```

The fact that the macro 'consider' destroys the accumulator makes it necessary for the programmer to include the additional '20 reserve' instruction in order to preserve the contents before specifying the element of the block required for storage. 'reserve' is an identifier naming location 18 which is used by SPAN for the operation of the macro 'store' i.e. 'store' places the contents of 'reserve' in the element specified.

Since faster macros are provided for single and two-dimensional arrays the main use of this method of handling array elements is for arrays of more than two dimensions e.g. the operations below on the four-dimensional array MATRIX.

```
                                              Comments
20 reserve                                    store accumulator in 'reserve'
consider[< + MATRIX >]
row  [< + 3>]
row  [< + 5>]
row  [< + 2>]
row  [< + 8>]                                 element MATRIX 3, 5, 2, 8
store                                         place contents of 'reserve' in this element
consider [< + MATRIX>]
row  [< + 3>]
row  [< + 5>]
row  [< + 2>]
row  [< + 7>]                                 element MATRIX 3, 5, 2, 7
take                                          bring element to accumulator
```

It should be noted that the maximum number of row macros in succession is six.

7.3 The selection of each element of a two-dimensional array takes about twice as long as that of a single-dimensional array; and selection of elements of arrays of more dimensions takes very much longer still. It is often possible to process an array one row at a time; when this is so, it is advantageous to detach each row from the array and process it as an array of lesser dimension, and finally to re-attach it to the original array prior to repeating the process on the next row. Apart from a gain in efficiency, this may also lead to a simplification of the program.

The required effect may be obtained by transferring in turn the codeword for each row of the original array from its place in an array of codewords into a fixed location which has a symbolic name and has been declared in an <u>array</u> declaration. The row is now processed under this name, and afterwards, the transfer is made in the opposite direction.

The macros 'detach' and 'insert' can be used for this purpose.

```
consider [< + A >]
row [< + 2>]
detach
```

places a copy of the codeword specified (element 2 of the first codeword block of A) in the codeword accumulator, CWACC, and sets the location which held the codeword to zero.

Considering the example in 7.2 to be part of a routine operating on the row 3, 5, 2 of the array MATRIX the same effect could be achieved by writing

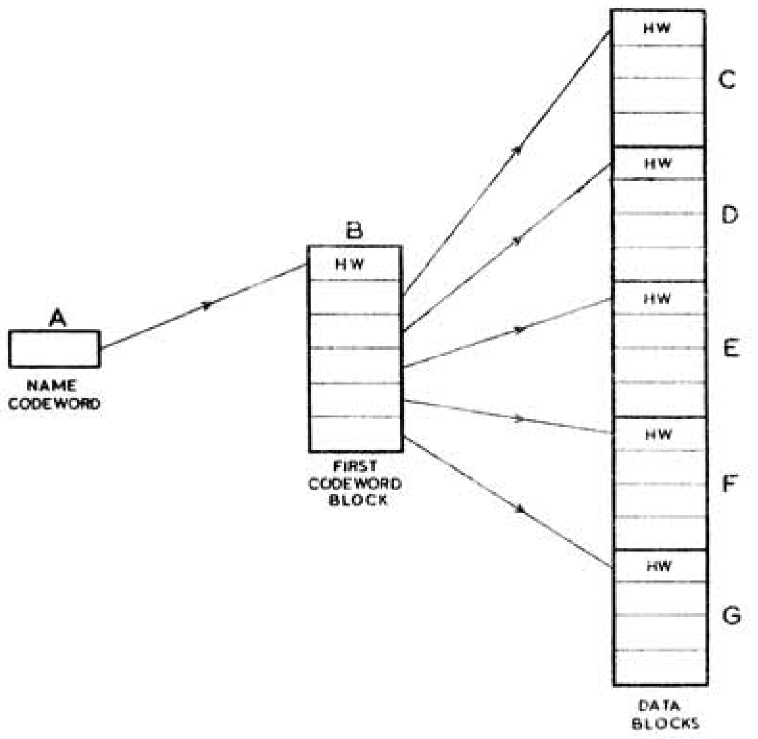
```
                                        Comments
20 reserve                              store accumulator in 'reserve'
consider [< + MATRIX>]                  array name MATRIX
row[< + 3>]
row[< + 5>]
row[< + 2>]                             select codeword for row 3, 5, 2.
detach                                  store codeword in CWACC
consider [< + TFT>]                     array name TFT
insert
30 reserve
st 1 [TFT,< + 8>]                       place the contents of 'reserve' in element 8 of TFT
tk 1 [TFT, < + 7>]                      bring to accumulator element 7
.
.
(rest of routine)
.
.
consider [< + TFT>]
detach                                  store codeword for TFT in CWACC
consider [< + MATRIX>]
row[< + 3>]
row[< + 5>]
row[< + 2>]                             store CWACC in element 3, 5, 2 of array MATRIX
insert                                  i.e. re-attach to original array.
```

Thus 'insert' copies the contents of CWACC into a named location and clears CWACC; 'detach' places a copy of the codeword in the specified position in the array structure, in CWACC and clears the original codeword. This ensures that a block in store cannot be referenced by more than one codeword.

7.4 As was shown by the example in 6.2, the macro 'ALLOC' provides a single block only, and arrays are allocated by building up a tree structure of linked blocks and using the 'consider', 'row' and 'insert' macros to store the codewords in the correct positions in the codeword blocks.

The 'insert' and 'length' macros operate on the block directly pointed to by the codeword specified and on no subsequent blocks linked in an array structure. Conversely, the 'delete' and 'detach' macros operate on all subsequent blocks linked in an array structure.

e.g. The operations on the two-dimensional array A (5 × 3) shown diagrammatically below have the following effects.



macros	*effect*
consider [< + A>] length	the length of block B is placed in the accumulator (= 5)
consider [< + A>] row [< + 4>] length	the length of block F is placed in the accumulator (= 3)
consider [< + A>] delete	blocks A, B, C, D, E, F and G are deleted. All codewords are replaced by zeros.
consider [< + A>] row [< + 4>] delete	block F is deleted and zero is placed in the 4th element of block B. A, C, D, E, G and other codewords in B are unaffected.

8. ADVANCE PLANNING OF STORAGE CONTROL

SPAN is written so that when data is referred to by a program it is brought into the main store if it is not already present. The scheme is, however, limited because it cannot look ahead in a program to see what data is to be needed next, so that if maximum speed of operation is required the programmer must do the necessary advance planning. By using special SPAN macros, the programmer can ensure that blocks that have been moved out of main store are brought back before they are needed by the program. Setting the value of the 'position' parameter of 'ALLOC' specifies the initial type of storage needed for the block: the following macros are for use once the block has been allocated.

All the macros described in this chapter have the same range of effect as the 'delete' and 'detach' macros operating on all subsequent blocks linked in an array structure.

8.1 Core Store Macros

To ensure that a block is placed in main store when it may be in either main store or core backing store write

```
consider [< + A>]
fast
```

This transfers, if necessary, the block or array named A and alters the status marker in the header word (see detailed description of contents of header word). Alternatively, the block or array can be moved to core backing store by

```
consider [< + A>]
slow
```

There is another pair of macros which do not have the effect of moving a block – 'fixed' and 'free'. Thus, before these are used, any positioning should be carried out by other means. These particular macros will normally only be used by systems control programs and have effect as follows

```
consider [< + A>]
fixed
```

This sets the position parameter to the value 3 so that the block or array A will not be moved by any storage collapsing (see APPENDIX 11) which may be required to find space.

```
consider [< + A>]
free
```

This sets the position parameter to the value 0 for the block or array A thus nullifying the effect of 'fixed'.

8.2 Magnetic Tape Macros

If the programmer knows that a shortage of space is likely, it is possible to make space available by writing some of the information held in main store or core backing store on to magnetic tape. This is particularly effective if it is known that the information being written away will not be required for a reasonable period afterwards. The macro which performs the operation is written as

```
consider [< + A>]
banish [HANDLER]
```

and the named block or array A is immediately copied on to the tape handler whose number is the integer held in the location with mnemonic name HANDLER. The block or array is then removed from core store and a reference is left to the magnetic tape copy in the name codeword for the block or array.

When an array is banished all related codeword blocks are copied to tape automatically, with the exception of the main codeword.

The actual operation of writing a block to tape is fully timeshared with background program operation but the foreground program is held up until it is completed.

It is sometimes useful to be able to specify that a block will not be used for a considerable period without causing the block necessarily to be written to tape. This may be done by setting a special bit in the header word of the block. This then indicates to the system that the block should be written to magnetic tape if the space it occupies is required for another purpose. The macros

```
consider [< + A>]
retire
```

are provided for this operation and set up the retire marker in the header word of the block or blocks of array A.

In order to bring back into the main store data previously written to magnetic tape write

```
consider [< + A>]
recall
```

Any necessary searching of the tape and transfer operations associated with the recall are timeshared with the background programs.

If the programmer fails to give a 'recall' instruction before referring to an element of a block of programs currently held on magnetic tape, the 'recall' is automatically inserted by SPAN.

A 'recall' macro may be given when a block has been acted upon by the 'retire' macro and the programmer may not know whether the block is held in core store or on magnetic tape. In this case, if the block is still in main store, the 'retire' marker is deleted and no further action takes place; if, however, it has been written to tape, space is found in the main store and the block contents are copied into it.

A program may use the magnetic tape macros of SPAN without specifying the handler to be used. This may be done by writing 'banish [0]' and SPAN will automatically use the handler which has been selected as <u>number 8</u> by the operator.

At this point, SPAN will also set its own origin from which to begin its block reference count for the magnetic tape alias of a block whose contents it may write away. This origin will be set equal to the current value of 'block max' for that handler, which is kept by PCP to indicate the number of blocks, already written on that tape. If the programmer wishes to set this origin, he must do so by altering the contents of 'block max'.

Should a programmer wish to 'banish' information to a specific handler (i.e. one other than the standard handler), the handler must have been previously booked through PCP. He must first set the SPAN origin for that handler by using the macro.

```
set origin [H,ORIGIN]
```

where H contains the handler number and ORIGIN contains the number of blocks already written on that tape. (H may not refer to the standard handler).

By setting ORIGIN to a value less than the 'block max' indication for that handler the user may cause unwanted information to be overwritten.

8.3 'read only' and 'write also'

Normally, when a block has been banished and recalled no record is kept of where the block was held on tape. If, therefore, the block is banished for a second time, a new copy has to be made on tape. Furthermore, owing to the characteristics of the tape, it is impractical to reclaim the space occupied by old copies and may cause a considerable length of tape to be used.

If the values of one or more elements of the block have been changed between successive banishments of the block, then the making of a new copy would be essential anyway. If, however, all elements have remained unaltered during their stay in core storage, the making of the second copy can be avoided. In order to do this the macro 'read only' must be given before the block is banished, i.e.

```
consider [< + A >]
read only
```

causes a marker to be set in the header word of A which indicates that the contents of the block (or array) are to remain unaltered and once the block is written to tape, further 'banish' macros do not cause rewriting but merely adjust the codeword to point to the copy already present on tape.

When a 'banish' is encountered referring to a 'read only' block, it is written to tape in the normal way, but when the block is recalled the tape address where the block was written (i.e. the alias) is held permanently in the top half of the header word of the block. On a further 'banish' instruction SPAN recognises the existence of the alias and does not copy the block to tape but merely adjusts the codeword to point to the previous copy on magnetic tape and deletes the block from the store.

The most important use of 'read only' blocks is for holding program which is not changed throughout the running.

If at some later stage in the program it is necessary to alter the values of a block which is declared as 'read only' the macro 'write also' must be given as

```
consider [< + A >]
write also
```

This cancels the effect of the 'read only' by removing the marker in the header word.

A subsequent 'read only' can be given, if required, and again further repeated rewritings on magnetic tape would be prevented.

'read only' and 'write also' may be used by the programmer in conjunction with core backing store with a similar effect. When a block is brought into main store from backing store; the 'read only' marker is examined and if set, the core backing store address (i.e. the alias) is stored in the header word of the block.

The copy of the block in core backing store is kept unaltered and if it is required at a later stage to transfer the main store block to backing store again SPAN recognises the alias in the header word. The block is then merely deleted from the main store and the codeword adjusted to point to the copy already in backing store.

9. ADDITIONAL FACILITIES

9.1 The programmer when writing a subroutine may wish to discover the properties of codewords used as parameters of his subroutine.

The least permissible subscript reference to a block can be discovered by means of

```
consider [< + A >]
lowbound
```

which places the lowest subscript reference i.e. the value of the index appropriate to the first data word (the next word after the header word) of the block in the accumulator. Consequently, the highest subscript which may be used is equal to (lowbound + length – 1).

Once a block has been allocated a need may arise to add or subtract locations from either end of it. This can be done by the macros

```
consider [< + A >]
newbounds [L, U]
```

'newbounds' provides a new block with upper and lower subscript range limits as specified by the integers held in U and L respectively, and copies over the common elements.

Thus in changing the subscript bounds any elements common to the old and new blocks retain their values, and any new elements are undefined.

e.g. if a block A originally held $a_{-5}\ a_{-4}\ a_{-3}\ a_{-2}\ a_{-1}\ a_0\ a_1$
and after held $a_0\ a_1\ a_2\ a_3$

the elements a_0 and a_1 are copied and a_2 and a_3 are undefined. Note that 'newbounds' cannot be used for fixed blocks as space may not be available if the new bounds lie outside the old ones.

The macros 'newbounds' and 'lowbound' have the same range of effect as the macro 'length', operating only on the block directly pointed to by the codeword specified. This facility is most useful for modifying the lengths of individual rows of an array.

The following example written in Algol shows how a two-dimensional array can be turned into a lower triangular array.

```
array    A [1 : n, 1 : n];
integer  i;
for i :  1 step 1 until n do
begin
         consider (A);
         row (i);
         newbounds (one, i)
end;
```

9.2 Table Handling

Sometimes the array facilities are not needed in the form outlined so far; instead numbers are to be stored, one at a time in order, with no particular need for use of a subscript. In this case a 'file' of numbers can be made which can be accessed quickly for storage and removal of numbers.

Each file is given a name codeword on allocation. A pointer in the codeword is automatically stepped on each time a number is filed, to point to the next storage location; and the extraction macro similarly causes the file position pointer to be moved back one. Therefore the files may be used either in a conventional manner or as a push-down list. Also there are two versions of each operation, one which applies to tables of ordinary values and one which applies to tables of codewords.

The macros intended for use with tables of numbers are as follows:

9.2.1 A file is opened by means of

setfile [A, size]

where size contains an integer n, the length of file required plus one location for the header word. This allocates a block of n locations and places the codeword in the location called A. Note that the length of the file as allocated initially is automatically increased when the file is in use if it becomes necessary. Also if size contains zero or one, a block of 5 locations is allocated, for the integers 2 upwards the number of locations specified is provided.

9.2.2 A number is filed from the accumulator by means of

file [A]

The operation of filing contains a test that there is room in the table for each new element and if not, the size of the table is automatically extended. SPAN does this by allocating a new block if necessary and transferring the contents. The position pointer in the codeword for the file is moved on one each time a number is filed.

9.2.3 The last number filed in A may be retrieved and placed in the accumulator by

popup [A]

which, at the same time, moves the pointer back one place. The location thus freed remains in the file for possible re-use, later. The 'popup' macro may be written repeatedly with no intervening 'file' macros, as long as there are numbers in the table before the pointer.

If there is no number in the table before the pointer then the pointer is not moved back and zero is placed in the accumulator.

9.2.4 When a file is not to be used for some time the space beyond the pointer can be reclaimed for SPAN by

chopfile [A]

The use of 'chopfile' in no way prevents further use of 'file' or 'popup' macros, but any 'file' macros leading to an increase in the new overall length of the table results in a call for more space at once and the creation of a new block when the space has been found.

9.2.5 The operations given below operate with codewords and the codeword accumulator, CWACC. The need for making files of codewords will usually arise in the writing of systems programs such as SAP, ALGOL, etc. Briefly the macros are:

cwsetfile [A, size]	which corresponds to setfile, and allocates a file for codewords of name A and length as specified by the content of location 'size'.
cwfile [A]	files a codeword from CWACC in file A and adjusts the position pointer.
cwpopup [A]	picks up the last word filed in A and transfers it to CWACC. The position pointer is moved back one.
cwchopfile [A]	removes surplus space from the codeword file A.

9.2.6 A file may be considered as a single dimensional array with a special index pointer. The programmer may use any of the macros described for use on blocks, for handling files, although care must be taken when using the macros tk 1 and st 1, since the subscript values of the elements of a file are changed whenever the pointer is moved i.e. whenever a number is filed or extracted.

The subscripts of the elements of a file can, however, be obtained by using the macro 'lowbound' and the relationship,

number of entries before pointer = 1 − lowbound

The pointer points to the position in which to file the next word.

10. CONTENTS OF A CODEWORD

The forms of codeword and information content are summarised in the table below. The meanings to be attached to the letters are:

s	256 + offset
n	size i.e. length of block + header word
h	magnetic tape handler number
r	record number on magnetic tape
a	offset + header word address

Codeword types:

10 s : 00 a	main store block
40 s : a	backing store block
42 s : 00 a	main store block tagged
42s / a	backing store block tagged
44 n : 0h r	magnetic tape block

The F1 digits of the codeword, with the exception of bits 38 and 34, indicate the type of storage in which the block is held. Bit 38 is used by the Binary Input/Output Program to mark blocks which are to be output. It is ignored by SPAN and the allocation routines set it to zero: assemblers and compilers set it to one in program, main codeword and constants blocks. Bit 34 is used by PCP in main store blocks to mark a buffer which has been obtained from SPAN by PCP.

'a' can occupy digit positions 1 to 19 in the codewords for backing store blocks for the reason that the header word address of backing store blocks can exceed 8191.

When blocks are moved from one type of storage to another by SPAN macros or space finding routines, SPAN automatically adjusts their codewords so that they always point to the correct place.

11. CONTENTS OF A HEADER WORD

The forms of header word used are summarised in the table below using the name notation as above as far as possible

00 0 : 00 n	free block
40 s B xy n	normal block
5h r B yx n	block with magnetic tape alias
6 b B xy n 7	block with backing store alias

The other symbols have the following meanings:

b — This represents the backing store address of the alias.

B (bit 20) — This bit holds the 'retire' marker:

if the marker is not set B = 0
if the marker is set B = 1

and the block will be written to magnetic tape if space is required.

x (bit 19) — This bit holds the 'check subscripts' marker:

if the marker is not set bit 19 = 0
if the marker is set bit 19 = 1

and a check of the subscripts is made on each reference to the block. (The block also has a tagged codeword.)

(bits 18, 17) — The content of the block is indicated by these bits as follows:

value	*content*
1	real numbers (floating point)
2	program
3	codewords
0	others e.g. integers

y (bit 16) — This bit holds the 'read only' marker:

if the marker is not set bit 16 = 0
if the marker is set bit 16 = 1

(bits 15, 14) — The type of storage is indicated by these bits (except in the case of magnetic tape blocks when this value has no meaning). The only possible values are:

value	*storage type*
1	main store
2	backing store
3	fixed in main store
0	main store or backing store.

The blocks with a magnetic tape alias hold the tape address i.e. the record number and handler number, where the block was written, in the first half of the header word. Similarly blocks with a core backing store alias hold the backing store address (b) of the copy in the first half of the header word. Since b can occupy bit positions 21 to 37, the first F1 digit is affected, and may be either 6 or 7.

When the 'retire', 'read only' and 'check subscripts' markers are set for an array, the header word is set in each individual block of codewords and data.

11.1 Allocation of blocks when header word known

An alternative method of allocation is provided which in some cases is more convenient, since it takes less space in the user's program. It is, however, most commonly used in routines such as those in the ALGOL compiler.

In this method of allocation the header word is given as the parameter of the macro, 'alloc' e.g.

alloc [HEADER]

where HEADER is a location holding the header word required.

The replacement texts for 'ALLOC' and 'alloc' are different to take account of the different form of specification of parameters, but they use the same allocation subroutine to find a block of store.

An example of the use of 'alloc' is

alloc [< 40 256 : 10 22>]

which allocates a block of real numbers in either the main or core backing store of length 21 usable locations: the 'retire' marker 'read only' marker and 'check subscripts' marker are not set.

12. ERRORS

Error indications are gien if one of the following occurs:

1. Error in tree structure, i.e. the correspondence between codewords and header words of allocated blocks has got out of step.

2. Not enough space of the type required.

3. An attempt to store data in a 'read only' block.

4. Subscript out of range.

5. No codeword in 'row' reference (liable to occur when a row of a multi-dimensional matrix has been detached or deleted).

13. FURTHER NOTES

1. All blocks used by SPAN must have their mnemonic names introduced in the SAC program by the control name array
2. Note that there are indirect equivalents to the macros tk 1 and st 1 (see page 8, issue 3). There are, however, NO indirect equivalents to tk 2 and st 2.
3. Note that the macro 'ALLOC' should now be written in capital letters and not as 'Alloc' as described in TIS.503.13.
4. If SPAN macros are written in a 503 ALGOL Mk. 2 program, the square brackets should be replaced by round brackets.
5. The effects of the following sequences of instructions should be noted.

```
          (i) The macros          consider [< + A>]
                                  consider [< + B>]
                                  consider [< + C>]
                                  retire
```

result in block (or array) C only being marked with retire. A 'retire' or similar macro must be repeated for each 'consider' i.e. for each block to be dealt with.

```
          (ii) The macros         consider [< + A>]
                                  SAP instruction(s) or SPAN macro(s) other than 'row'
                                  row [I]
                                  insert
```

A SPAN macro or SAC instruction other than 'row' may destroy the contents of registers set up by a 'consider' macro. Therefore the following macros must be immediately preceded by a 'consider' with, if necessary, a series of 'row' macros only in between:

insert, detach, take, store, delete, length, lowbound, newbounds, [L, U], fast, slow.

(iii) The macro 'length' can be used to determine the length of any block but account must be taken of array structures, i.e.

```
consider [< + A>]
length
```

if A is an array, will give the length of the first codeword block only.

14. SUMMARY TABLE

14.1 The following macros have explicit parameters and need no introduction by 'consider' and 'row':

ALLOC	[size, type, position, offset]	
alloc	[header]	
tk 1	[A, I]	tk 2 [A, I, J]
st 1	[A, I]	st 2 [A, I, J]
setfile	[A, size]	cwsetfile [A, size]
file	[A]	cwfile [A]
popup	[A]	cwpopup [A]
chopfile	[A]	cwchopfile [A]

14.2 The following macros have a parameter implied by calls of the type

```
                consider [< + A >]
and             row [I]
```

which come immediately before them.

insert	delete	detach
length		
lowbound	newbounds [L, U]	
read only	write also	
fast	slow	
fixed	free	
retire	banish [handler]	recall
take	store	

14.3 The 'check subscripts' marker may be set in the headerword by adding 4 to the parameter 'type' when using 'ALLOC', or by placing the marker in the headerword specified in an 'alloc' macro (see page 22 description). The effect is such that whenever reference is made to the block, the index of the element referred to is checked to ensure that it lies within the lower and upper subscript bounds. The codeword of the block will take the special 'tagged' form which it also takes when the 'retire' operation is carried out.

APPENDIX 1

Structure of a program block

Program words start in 1, of a program block (i.e. the word after that pointed to by the codeword). The word immediately after the header word is special and contains information for use by relocation routines.

The first pair of relocation bits refer to 1, of the block. There are no relocation bits for the special word.

APPENDIX 2

Search for free store

If the programmer specifies that a block must be allocated in the main store, or it is necessary for immediate program requirements to bring a block from auxiliary store into main store then SPAN enters a search routine and finds free store in the following manner. It examines the header words of the blocks in the main store to determine whether the block is free and if in this search two adjacent free blocks are found these are immediately amalgamated. If there is insufficient consecutive free storage in the main store the following action is taken:

If the block requested is a 'normal' block, a similar search is made of backing store, and if there is still insufficient space then a 'collapse' of main store takes place. This happens in the following manner:

Main store is first cleared of as many blocks as possible by writing blocks with the 'retire' marker set to magnetic tape and writing all others possible to backing store. The remaining blocks in main store that are moveable are then moved as far as possible towards the low addressed end of store.

SPAN will never move a codeword block out of main store, during a 'collapse', and the programmer must always make sure that codewords for every main store block are held in main store, either in a moveable or fixed block. All fixed blocks (position parameter set to 3 when allocated) are usually grouped together at the high addressed end of store by the allocation routine and no fixed block is ever moved when collapsing the store.

APPENDIX 3

This appendix contains details of additional facilities which have been included in SPAN together with details of the basic version of SPAN which has been produced for the reasons described below.

A list of the error messages which SPAN may display is also included, together with the meaning attached to each error message. No continuation is possible once a message of this type has been displayed.

Trace Facility

In the event of an error occurring, which the user attributes to SPAN for some reason, a trace facility may be invoked by setting certain keys of the word generator on the control console. The detail of this trace will assist diagnosis of the error should the user wish to supply details of the error to Elliotts or merely in order to provide a more comprehensive program trace (in addition to a store picture).

If Key 19 is depressed during a program run, the address of the label-pseudo codeword (see detail SAP2 description) of each 'enter', 'exist' or 'trans' obeyed is output on punch 2.

If Key 18 is depressed, each time a 'collapse' of store occurs 'CLAPSEE' is displayed on the typewriter. The last E is output on completion of the collapse so that the occurrence of a program loop inside 'collapse' itself may be readily observed.

Extra macro

The macro 'collapse' is now included. This will force a collapse of store and, on return to the main program, the size of the longest free block of main store is held in location 18 (reserve).

Restriction

It should be noted that a block on core backing store can be neither fixed nor banished. An error message to this effect will be output in each case.

Setting-up Entry

Before SPAN may be used in any way an entry must be made to SPAN in order to cause it to set up both main store and core backing store. At the same time the operator must inform SPAN how many units of core backing store are available by choosing the appropriate entry:

Size	Entry
1 unit	SPAN
2 units	SPAN;2.
3 units	SPAN;3.
4 units	SPAN;4.
No units	SPAN;5.

Error Indications

No continuation is possible after each of the messages below have been displayed:

MESSAGE	MEANING
SPAN ERROR	An error has been found in the tree structure of the store, e.g. zero headerword or meaningless codeword.
SPANERROR 1	No room in main store to meet space request, despite action of the 'collapse' routine.
SPANERROR 2	No room in core backing store.
SPANERROR 3	An attempt has been made to write into a block marked 'read only'.
SPANERROR 4	The subscript used in a 'row' macro is too small (out of range).
SPANERROR 5	As for error 4 but the subscript is too large.
SPANERROR 6	A zero codeword has been specified in a 'row' macro.
SPANERROR 7	Codeword of unknown form, e.g. Fl bits are 65, 66 or 67.
SPANERROR 8	Magnetic Tape codeword referred to in error.
SPANERROR 9	CBS not available (e.g. because of parity failure)
SPAN ERROR NBOFLO	Newbounds overflow, i.e. lower subscript is higher than the upper subscript.
SPANERROR MTBNTL	The number of the magnetic tape block number specified in a 'banish' instruction is too large.
CLAPSE ERROR	An unusual headerword found during a 'collapse'.

Two versions of SPAN

SPAN must always be the first program input and therefore occupies the low-numbered end of the store. Since it is coded in SAP binary it occupies a large, fixed block of store and thus restricts the size of the free area into which SAP2 may assemble a program, although SAP2 itself only makes use of the most basic facilities provided by SPAN. Due to a rapid build-up of the dictionaries required by SAP2 when assembling large systems programs such as PCP, the need has arisen to reduce the space occupied by SPAN at assembly time.

The assumption is made that all necessary programs are assembled to store before one of the programs is run. It follows that, at run-time, the space occupied by SAP2 may be claimed. This assumption makes it possible to use a restricted version of SPAN (code:span) at assembly time which contains only those facilities required by SAP2. At run time, both SAP2 and the restricted SPAN are replaced by the full version of SPAN. Before the assembly of a further set of SAP2 programs the SAP2 store configuration must be set up again.

In order that the exchange of programs may take place automatically the full version of SPAN is stored in the core-backing store (using the program SETCBS as described in 2.5.4 and newsletter P8) and is brought down to replace SAP2 and span by the action of the program SPANSWAP (which removes itself also). The space occupied by SPAN in core backing store is also released for use by the program at run-time.

The procedure for setting up the store as described above is found in detail in section 2.5.4 and newsletter P8. It is there recommended that the basic store configuration (and others) be held as a batch on magnetic tape where fitted. This provides for fast retrieval of the standard store configuration described above ready for the next program run.

503 TECHNICAL MANUAL

VOLUME 2: PROGRAMMING INFORMATION

PART 5: PROGRAMMING AND OPERATING AIDS FOR THE NON-BASIC 503

SECTION 3: PERIPHERAL CONTROL PROGRAM (PCP)

CONTENTS LIST

1. INTRODUCTION

1.1 PCP in the Non-Basic Software System

Although this section is concerned with a detailed description of the facilities in the Peripheral Control Program, this program is itself an integral part of the software system, so that it cannot be considered and certainly not used in strict isolation from the other programs. The PCP and the Storage Planning and Allocation (SPAN - 2.5.2.) are the most basic of the programming systems and SPAN, at least, must always be in store when running a program written in SAP Mk.2, ALGOL Mk.2 or FORTRAN $\overline{\underline{\text{IV}}}$ (all specified in Vol.2, Part 4); even though the programmer may not be using these programs directly, he may be using other systems programs or writing in a language, which automatically use the facilities of PCP and SPAN.

This section of the Manual is concerned only with details of how to use PCP directly, but it should be realised that the time-sharing facilities are in many cases being used automatically. The ALGOL Mk.2 and FORTRAN compilers incorporate the time-sharing and storage allocation facilities automatically.

1.2 General Scheme

A non-basic 503 computer configuration is one which includes one or several peripheral devices in addition to the basic tapę readers and punches, and typewriter. For the purposes of the non-basic software system, the minimum configuration for the full system is two tape units and a unit of core backing store.

The attachment of peripheral devices to a central processor, gives the system greatly increased potential and versatility, but this can only be fully realised with the aid of a time-sharing system.

Without some form of time-sharing any instruction from the program to a busy device causes the program to be held up until that instruction can be accepted. This both wastes central processor time and means that other devices which could well be in operation are idle. Instructions are carried out sequentially in strict order and cannot be "postponed" until they become acceptable. The programmer has some assistance in organising peripheral transfers from the Autonomous Data Transfer and the Interrupt Facilities (see 503 Manual 1.3.1 and 2.1.1.2 respectively), but it is by no means as easy programming task to use these to best advantage.

For this reason, Peripheral Control Program (PCP) is designed to administer a time-sharing scheme for non-basic configurations. It takes over from the programmer the task of arranging ADT's and Interrupts, so as to obtain maximum efficiency from all peripheral devices, without impairing the efficiency of the central processor.

Under the scheme, the program can set all peripheral devices in operation and concurrently be carrying out calculations in the central processor. If the program gives an instruction to a device already engaged on a previous operation (i.e. busy), the instruction is held over by the PCP until the device is free, thereby allowing the program to continue its own calculations and also keep other devices busy. If yet another instruction is given for the same device, a whole queue of tasks is formed and carried out in order as the device becomes free. Thus a peripheral device need never be idle nor a program kept waiting by a peripheral device.

The supervision of holding instructions for each device on a queue and handing over the instruction when the device becomes free is carried out quite automatically by PCP and requires no effort on the programmer's part.

The scheme outlined so far only refers to time-sharing of separate parts within a single program. However, some programs consist almost exclusively of input and/or output, and in these cases there would not be sufficient calculation to keep the central processor running concurrently with peripheral devices. Provision is made, therefore, to run such programs (known as background programs) independently yet concurrently with a main (foreground) program.

It may occur quite frequently that a foreground program cannot usefully continue until the input/output instruction just reached has been carried out. In this case the foreground program can wait until the essential transfer is complete and the background program take over control of the central processor.

The time-sharing scheme achieves maximum efficiency when all the peripheral devices are kept running continually at their maximum speeds and there is no hold up or lull in work for the central processor.

2. PROGRAMS TO BE TIME-SHARED

As outlined in the Introduction, there are two aspects of time-sharing incorporated in the overall scheme, in order to accomodate two differing yet complementary types of program.

In a user's main (foreground) program the input/output routines are regarded as semi-autonomous programs and are time-shared with that program's work for the central processor; once the initial order to a peripheral has been issued, the main program can continue running concurrently with the input/output routine. This scheme would achieve high efficiency if all programs had the characteristics of a foreground program. However, the scheme has been designed to include other programs in which there is little, if any, processing of data but rather conversion or transfer, of data; these are known as background programs. It would be extremely uneconomic to run such programs alone since the central processor would be largely unoccupied during the run. Therefore a background program can be time-shared with the foreground program. This does not mean that both programs may use the central processor simultaneously: only one program may be in control at any one time, but this may be either of the programs currently being time-shared. The allocation of central processor time to each program is dealt with automatically by the control section of PCP.

In practice most of the central processor time is claimed by the foreground program with only very short periods given over to the background program so that it may initiate further peripheral transfers.

No two programs to be time-shared in the same run may make use of the same peripheral devices; a device must be allocated exclusively to one program for that run. Very rarely, if at all, does a foreground program use all peripheral devices, so by judicious choice of the background program to be run with a particular foreground program, all devices may be in action during the run. Thus whilst background programs are a most uneconomic proposition if run alone, they do in fact increase the efficiency of a system by complementing a foreground program: they can fill up any central processor time caused by hold-up in a foreground program and employ any devices which would otherwise be idle.

2.1 Foreground Programs

A user's main program is classed as a foreground program; only one program of this type may be active in the store at any one time. This master program may control several slave (common) programs or subroutines, which can only be entered on an instruction from the master program. A slave program may only use those devices reserved for its master program.

A foreground program may use the full facilities of PCP. It may also make direct use of SPAN thereby having available the full range of dynamic storage allocation facilities. A slave program may use the same facilities as its master.

2.2 Background Programs

A background program is, as the name implies, a less important program, supporting yet independent of the main program. It is very often in the nature of a pseudo off-line conversion program e.g. printing up data from magnetic tape to the line printer, transferring data to or from punched cards, etc.

So that it does not impinge in any way on the foreground program, a background program has the restriction that it may not make direct use of SPAN. PCP implements some of the facilities of SPAN in order to obtain and return space dynamically for the programs under its control; this is permitted for background programs, as being indirect usage, on condition that the special entries to PCP for buffer allocation and deletion are used (see 9.3.3).

The ALGOL and FORTRAN compilers make extensive use of facilities not available to background programs, which are, therefore, always written in the SAP 2 programming language.

3. FORM OF DATA FOR PCP

Data to be input or output by means of PCP macro instructions must be presented in one of two forms: either binary wholewords or alphanumeric characters. It should be noted that PCP itself does not contain any routines for converting numeric characters into binary wholewords and vice versa.

If output takes place in binary wholewords, it is assumed that the information is intended for re-input in a subsequent run; this form is not valid for the line printer or the typewriter since the print-up produced would be unintelligible. In character format the data is intended for eventual, if not immediate printing. For example, output from a foreground program in character form to be printed on the line printer, may first be output to magnetic tape as being a much faster device; the subsequent printing from magnetic tape to line printer later forms a typical background program.

4. BUFFERS

All transfers of data to and from the computer takes place via buffer areas: a buffer is any area of store in which data is held as an intermediate stage during transfer between devices, and which compensates either for the difference in the speed of the two devices, or for the time lag between two events. (In this context the main store is also considered as a device.)

An area of store which is to be used as a buffer must be allocated as such, either by the programmer himself or by PCP. In either case, the locations are obtained from and constructed to function within the SPAN system. Each buffer forms a separate SPAN block and as such has a codeword associated with it, held in a SPAN codeword block; the codeword is a pointer to the position of the buffer, holding in bits 1-13 the address of the first word of the buffer. Reference to a buffer is always made via the address of its codeword. The first word of a buffer is a headerword containing in bits 1-13 the size of the buffer.

For input the buffer may be obtained by the programmer using SPAN macros directly or automatically by PCP from SPAN whenever a transfer is requested. In the latter case, the programmer does not know the position of the buffer. If he wishes to have close control over the position of a buffer, he would obviously allocate the area himself, but if this is of little consequence, he would do better to leave it to PCP; the advantages of this are that the area is not reserved until it is actually required and SPAN can then choose the best position for it at that moment. If the buffer is supplied in this way, then on execution of the transfer instruction the codeword of the buffer is in bits 1-13 of the accumulator. All bits should be preserved for any future reference to that buffer.

On input it is not usually necessary for the programmer to know where the data will be placed, but for output the codeword of the buffer must always be specified, so that the intended data is output. The same buffer area can be filled and/or emptied over and over again but whenever a buffer becomes redundant it should be returned to SPAN for re-use.

5. <u>QUEUEING</u>

The first time in a program that transfer to any peripheral is requested, that transfer will be carried out immediately, since the device is reserved for that program as being free. However, if a transfer is requested for the same device before the previous operation is completed, then the request is normally held over by PCP until the device is free, whilst the program is able to continue. When the device is available again, it informs PCP which then hands over the next operation to be performed. In this way, PCP can build up a queue of operations for each device, adding and removing requests to and from the queue whenever they are made by the program or executed by the device. Each entry on the queue contains exact details of the buffer concerned and the function to be performed; all requests are dealt with strictly in the order in which they appear in the queue, except with controlling functions for magnetic tape handlers (Chapter 20).

In some cases the calling program may be unable to continue until the transfer has been carried out. It is, therefore, possible to specify that the program must wait until the data has been input/output. Control is then transferred to the other active program in store which can make use of the central processor or failing this a wait ensues until the essential transfer is completed.

When time-sharing methods are not being used, instructions to input/output data normally occur at the beginning/end of a program, with little peripheral activity in the middle. In order to gain full advantages from PCP, this bunching of input and output should be avoided. Peripheral transfers should, as far as possible, be spread evenly throughout the program, so that the length of queue for a device is reasonably constant.

6. METHOD OF CONTROL

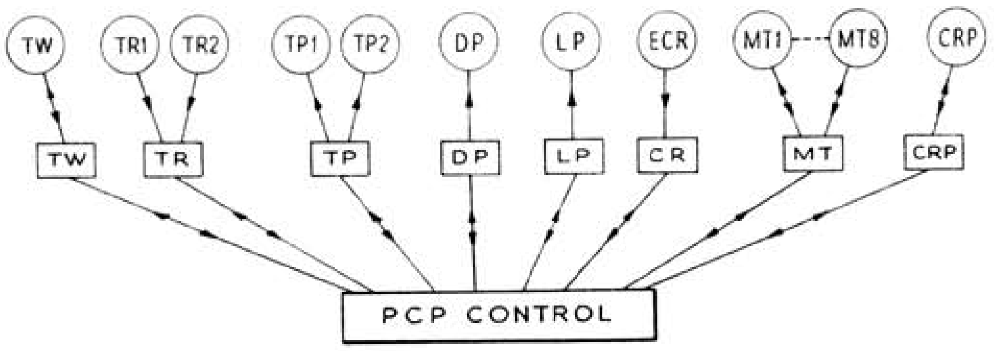
PCP falls structurally into two distinct sections; the core of the program is the control section which handles the administration for all transfers to peripheral devices, shares out central processor time to the active programs and holds information on the current state of these programs.

The other section consists of device routines, one routine for each device to be controlled. As further devices become available, the relevant routine will be added to PCP. In addition to carrying out the physical input/output, of data, the routines check, where possible, accurate functioning of each device before using it and, in the case of magnetic tape, contain re-read facilities for parity failures etc.

Information on the current state of each program is held in the Control List of PCP. Since PCP has details of each program, it can transfer control to the background program when, and if, the main program is held up. The program or device routine currently in control is indicated by the Current Program Pointer.

The administration of queues is handled automatically by the control section of PCP. Whenever a device becomes free it sends through an interrupt to PCP indicating that it is awaiting further instructions. PCP then directs the next instruction on the queue to the appropriate device routine. If the queue is empty, the device remains idle until a further request occurs in its calling program.

The following diagram shows the lines of communication and flow of data between PCP control section, the device routines and the devices.



TW - Typewriter	DP = Digital Plotter	MT - Magnetic Tape
TR - Tape Reader	LP = Line Printer	CRP = Card Reader/Punch
TP - Tape Punch	ECR = Elliott Card Reader	

Note that ▭ *indicates a program and* ○ *indicates a device*

Communication between PCP control and device routines is two-way: from device routine to PCP whenever a device interrupts to say it is free, from PCP to routine to re-direct peripheral transfers.

Between device and device routine data flows according to whether it is for input and/or output.

7. MACRO INSTRUCTIONS

All instructions in the user's programs to the PCP are in the form of macro-instructions i.e. one instruction which is replaced by many when the program is translated; it consists of one or more words followed by a set of parameters in square brackets. The macro is written following the conventions of Symbolic Assembly Code. Each parameter of the macro is normally an identifier which specifies the location holding the actual value of the parameter; this means that the effect of a particular macro can be changed by substituting a different value in the parameter location rather than by re-writing the actual macro.

7.1 Data Transfer Macro

Each macro concerned with filling, emptying and transferring buffers is made up of the word "buffer" followed by the parameters in square brackets. However, since only the first six letters are significant, the name "buffer" may be qualified to distinguish its function, e.g.

```
buffer in     [             ]
buffer out    [             ]
buffer fill   [             ]
buffer empty  [             ]
```

A standard list of parameters is provided to make the macro as general as possible, but only the relevant paameters for each device need be included, as explained below.

The full parameter list is:

D	Device Class
N	Device Number
B	Buffer Specification
F	Function
EL	Error Label
M	Mode
C	Special Purpose
K	Special Purpose

The parameters must appear in the order specified above. Where certain of the parameters are not applicable, its position must be occupied by the digit zero. Trailing null parameters may <u>not</u> be omitted. Note that the parameters are not absolute values, but identifiers for the location holding the value.

Since the parameter is merely an identifier for a location, it is not necessary to use the mnemonics given in the above list; the identifier may be any meaningful name, but they must, of course, appear in the correct order.

The permitted ranges of values for each parameter and their meanings for each device are specified in chapters 13-20.

<u>N.B.</u> The error label must be one which has been declared globally and must at all times be quoted in the form:

<error label> * <block name>

7.2 Example

This short section of program would cause a buffer of standard size to be input from reader 1 and output on punch 2.

```
   .
   .
   .
   .

26      device                   (set device = 0, i.e. tape readers)
30      <+1>
20      number                   (set device number = 1)
30      <+2>
20      function                 (set function = 2, i.e. provide)
26      mode                     (set mode = 0, i.e. character)
buffer in [ device, number, 0, function,    (PCP to provide buffer)
                                 E*B, mode, 0, 0 ]
20      BUFFER                   (store address of codeword for buffer
                                 just input, in location BUFFER)
30      <+1>
20      device                   (set device = 1, i.e. tape punches)
20      function                 (set function = 1, i.e. output)
30      <+2>
20      number                   (set number = 2)
buffer out [ device, number, BUFFER, function,
                                 E*B, mode, 0, 0 ] (output to punch 2)

        .
        .
        .
        .
```

The program now continues although the buffer will not have been output completely.

Although the above example uses parameter identifiers, it is also permissible to write the absolute values required for that macro, in the parameter list; this means, however, that to change the values of the parameters, the macro must be rewritten. Using absolute parameters the above example could be written:

```
buffer in [ 0, <+1>, 0, <+2>, E*B, 0,0,0 ]
20      BUFFER
buffer out [ <+1>, <+1>, BUFFER, <+1>, E*B, 0,0,0 ]
```

7.3 Device Macros

Since a device may be used by only one program at any one time, the program must ascertain whether the required device is available (i.e. not reserved for any other current program) and if so, reserve it. This is done by using the following macro in the calling program (the parameters have the same significance as in the macro "buffer"):

```
open device [ D,N,EL ]
```

which reserves device D (N) for that calling program. If the device is unavailable, exit is made to the error label.

Before the end of each program, the instruction

```
close device [ D,N,EL ]
```

should be given for each device used by that program. This causes all output requests on the queue for the device to be carried out in turn and all input queues, if any, to be cancelled; the calling program is not re-entered until the output queue for that device is empty. Failure to give the close device instruction may cause output to be incomplete.

The instruction

cancel device [D,N,EL]

has more immediate effects than the "close" macro. It causes all requests on the queue for that device to be cancelled including those partially obeyed, and should, therefore, be used with caution.

There is a further device macro

shut device [D,N,EL]

which simply causes the calling program to wait until all the instructions already on the queue at that time have been executed. The device is still available for that program since the device remains booked to it until either a "close device" or "cancel device" instruction is issued.

PARAMETERS AND DEVICES

The following chapters (8-12) explain the purpose of each parameter in the macro instructions insofar as this is common to all devices. Where the meaning of a parameter is peculiar to a particular device, full details are given in the chapter for the device concerned. The actual values to be used in setting a parameter to achieve the required effect are given in the chapter for each device (13-20).

As and when further peripheral devices are available for inclusion in a 503 computer configuration, the relevant device chapter (21,....) will be added to correspond with the device routine incorporated in PCP.

8. DEVICES

8.1 Device Classes (D)

The number assigned to each class of device administered by PCP is as follows:

D = 0	Tape Readers
1	Tape Punches *
2	Line Printer
3	Elliott Card Reader
4	Card Reader/Punch
5	Digital Plotter
6	Control Typewriter
7	Magnetic Tapes

These numbers are used to set parameter D of the 'buffer' macro.

8.2 Device Number (N)

If there is more than one device in a class, the parameter N is used to specify the device concerned. This will be relevant, for example, for the tape readers and tape punches (1 or 2), and the magnetic tape handlers (1 up to 8). Where there is only one device in a class, this parameter is irrelevant.

9. ALLOCATION OF BUFFERS (B)

Buffer areas are always obtained from SPAN, either directly by the programmer, or via the PCP Buffer Allocator. After using the buffer the programmer must always ensure that it is returned to SPAN for further use.

9.1 Storage Control

When an input request to 'prepare' a buffer via the PCP Buffer Allocator appears at the top of a queue, PCP requests an area from SPAN. If, however, SPAN is unable to provide a buffer of the required size without collapsing the store (i.e. moving everything in store closer together so that the required number of adjacent locations can be provided), then no action is taken at that moment. When the queue for that device is re-examined, a further attempt to obtain space is made. If space is still unavailable when the 'prepare' instruction becomes 'provide', the request is repeated until space becomes available due either to the fact that an output queue is exhausted, thus releasing space, or to the recognition by PCP that a collapse must be enforced.

9.2 PCP Buffer Allocator

Buffer areas for the input and/or output of data may be obtained from SPAN by the PCP Buffer Allocator. The advantage in obtaining buffers via PCP is that the space is not allocated until it is actually required; the programmer does not know the position of the buffer until the buffer specification (address of codeword) is placed in the accumulator once the full buffer has been provided. The specification must be stored immediately and any reference to that buffer or any of the elements in it must always be made via its codeword.

When PCP obtains a buffer for input, although the area's position is unspecified, the size of the area may be specified. If it is not, then a standard size buffer of 121 words is provided, unless buffers for that device are of some other standard size (e.g. 80 words for card equipment). For output the size is implicit in the buffer specification.

2.5.3

Buffers for input may be obtained by the macro

'buffer in' with parameter B = 0

After input, the buffer area must either be used subsequently for output or returned to SPAN for use. It may be returned by using the macro:

BRETURN [Buffer Specification]

For output, either a buffer previously obtained for input is used, or a new buffer obtained via PCP, thus:

BSUPPLY [<+N>]

would provide a buffer of size N, to be filled and output. <u>All buffers obtained via PCP are automatically returned to SPAN as soon as output is complete</u>. * 8.3

The methods for obtaining and disposing of buffers via PCP can be shown as follows:

(a)

```
buffer in  [    with B = 0      ]
20      Buffer Specification
process buffer
        :
```

$$\begin{tikzcd} & \text{process buffer} \arrow[dl] \arrow[dr] & \\ \text{output [with B = Buffer Specification]} & & \text{BRETURN [Buffer Specification]} \end{tikzcd}$$

(b)

```
BSUPPLY [ <+N> ]
20      Buffer Specification
fill    B
        :
buffer out   [     with B = Buffer Specification    ]
```

9.2.1 Buffer Specification

After a buffer has been obtained via PCP, the accumulator holds the 13-bit buffer specification: the address of the SPAN codeword for the buffer. If on exit from the subroutine which supplies the buffer, the accumulator is empty, then it is not possible to provide a buffer of that size without collapsing the store. The procedure taken is described in section 9.1 above

Any further communication with PCP concerning the buffer must be made via the buffer specification. The SPAN macros Tkl and Stl (described fully in 2.5.2., 503 Manual) may be used to deal with the elements of the buffer. For example, the instruction to bring to the accumulator, the first word of a buffer whose specification is stored in location BUFFER is:

```
Tkl [ BUFFER, <+1> ]
```

9.2.2 Example

In the following example a buffer is input, its first word tested and if negative, the buffer is output; otherwise the buffer area is returned to PCP and a further buffer input.

The macros are written with absolute parameter values:

```
input)    buffer in [<+0>, <+1>, 0, <+2>, E*block, 0,0,0 ]   (PCP to provide
                                                            buffer from reader 1)
          20      BUFFER                                    (store buffer
                                                            specification in
                                                            BUFFER)
          Tkl     [ BUFFER, <+1> ]                          (bring first word
                                                            to acc.)
          41      output                                    (test if negative)
```

```
        BRETURN        [ BUFFER ]                                       (return unwanted
                                                                        buffer to PCP)

        40    input                                                     (jump to input new
                                                                        buffer)

output) buffer out [ <+1>, <+2>, BUFFER, <+1>, E*block,0,0,0 ]  (output buffer to
                                                                   punch 2; the buffer
                                                                   area is auto-
                                                                   matically returned
                                                                   to PCP)
```

9.3 Own Buffers

When the programmer wishes to allocate his own buffers, this must be done using the macros of SPAN (a fuller description is given in SPAN, 2.5.2. - and in SAP Mk.II. 2.4.1). Although the space for 'own' buffers is not allocated until run time, at assembly time each buffer must have one location reserved for its codeword. This is done by writing the SAP 2 control word array followed by an identifier. Later in the program when the space is requested, the codeword for the buffer must be placed in the location specified by this identifier.

Space for the buffer area is reserved by means of the SPAN allocation macro. This consists of the mnemonic 'ALLOC' followed by four identifiers for the locations holding information on the size, type, storage status and offset of the required buffer, thus:

```
ALLOC [ Size, Type, Status, Offset ]
```

As with the macro 'buffer', the absolute values may be written in place of the identifiers.

When the ALLOC instruction has been carried out, the codeword for the buffer is in the accumulator and must then be stored in the reserved location by the pair of macros 'consider [X]' and 'insert'; these two macros have the effect of placing the content of the Accumulator in the location specified by the parameter of 'consider'.

9.3.1 Example

The full scheme for allocating "own" buffers is shown in the example below which allocates a buffer, named by the identifier "Mine", of 80 locations; inputs the data from paper tape, adds the numbers together and outputs the original numbers to cards.

```
              :
      array  Mine;
              :
              :
      begin  example;
              ALLOC [ <+81>, <+1>, <+1>, <+0> ]        (allocate 81 word buffer)
             consider [ <+Mine> ]
              insert
              buffer in [ <+0>, <+1>, <+Mine>, <+2>,E*B,0,0,0 ](input data from
                                                                 tape reader 1)
              26      sum
              02      0
              20      I
next) Tkl [ <+Mine>, I ]                               (pick up element I of buffer
                                                        add to total)
              24      sum
              32      I
              05      <+80>
              41      next
              buffer out [ <+4>, 0, <+Mine>, <+3>, E*B, <+2>, <+3>, 0 ]
                                                       (output numbers to card punch)
              consider [ <+Mine> ]                     (return the buffer area to
                                                        SPAN for re-use).
             delete
              :
end   example;
```

9.3.2 Undetectable Error in handling 'own' buffers

When a programmer is dealing with information held in buffers which he himself obtained from SPAN, then unless certain rules are followed, the correct data may not necessarily be found in a buffer at the exact time of input or output.

He must ensure that he does not attempt

(a) to output data which may not yet have been fully input.

(b) to use a buffer area which is still part of a queue.

For example, it is permissible for the instruction

"prepare to fill 'own' buffer from Tape Handler 1"

to be followed immediately by

"output same buffer to Tape Handler 1."

These instructions would be obeyed in the order in which they appear in the program and the correct information could be transferred. However, the following sequence of instructions,

"prepare 'own' buffer from Tape Handler 1"

followed by

"output same buffer to Tape Handler 2"

would not give correct results in the case where at the time of attaching the request to a queue, Handler 1 was busy but Handler 2 was unbusy; the input of data would be postponed until Handler 1 became unbusy, yet the contents of the specified buffer, whatever they may be at the time, would be output immediately to Handler 2. PCP will not detect that this may occur, but it is neverthe less regarded as a programming error.

9.4 Buffers for Background Programs

The allocation of buffers for background programs works under the same broad scheme as with foreground programs; buffers may either be obtained from SPAN or provided by the programmer. However, certain restrictions apply.

9.4.1 Buffers from SPAN

Although a background program may not use any of the storage control facilities directly, it may obtain space required for buffers from SPAN provided that it obtains them through PCP. However, this does not mean that parameter B in a buffer input instruction may be set to zero, as it is in a foreground program. Background programs must use the macros BSUPPLY and BRETURN to obtain and return buffers.

These two macros therefore provide the only method by which a background program can use SPAN.

Thus the instructions for obtaining from SPAN a buffer of, for example, 20 Locations for a background program are:

```
30      <+20>                 (store no. of locations required in SIZE)
20      SIZE
BSUPPLY [ SIZE ]              (provide buffer area)
20      BUFFSPEC              (store buffer's codeword address in
                              BUFFSPEC)
```

The codeword address for the buffer can then be used to set parameter B of the input "buffer" macro:

```
buffer in [ <+0>, <+1>, BUFFSPEC, <+2>, E*B, 0,0,0 ]
```

After input, the buffer area may either be used for output (in which case it is automatically returned to SPAN when empty) or it must be returned to SPAN by means of the macro:

BRETURN [BUFFSPEC]

9.4.2 Own Buffers

If the programmer wishes to allocate his own buffer areas, he cannot do this in the normal way, since this involves direct use of the SPAN macros, ALLOC, consider, etc. Instead he must construct the same scheme for buffers as would be obtained automatically by SPAN: this entails reserving the necessary fixed space, composing the codeword and headerword, and storing them in the appropriate locations. Thus PCP accepts a buffer for queueing under the same conditions as if obtained from SPAN, except that the buffer cannot be deleted dynamically after use. The contents and meanings of headerwords and codewords are given in detail in Chapters 10 and 11 of 2.5.2 (Storage Planning and Allocation).

The following example shows the method of allocating 'own' buffers in background programs:

Instruction		Meaning
<u>array</u>:	holdcode;	reserve one location for buffer's codeword
<u>data</u>	B (20);	reserve 20 locations for buffer + 1 location for headerword
30	<10 256 : 00 B >	Form codeword in Accumulator
20	holdcode	Store in reserved location
30	<40 256 : 11 21>	Form headerword in Accumulator
20	B	Store in 1st location of buffer
buffer in	[<+0>, <+1>, <+holdcode>, <+2>, E*B, 0,0,0]	Use codeword's address to set parameter B

10. FUNCTIONS (F)

Several alternatives for both standard input and output functions are provided in order to accommodate both methods of time-sharing and to cover as many of the peripheral devices as possible. However, as a result of their peculiar functions, certain peripheral devices (e.g. the card reader/punch, the typewriter) either cannot be included under the general scheme of standard input/output functions, or require extensions to the standard functions. Therefore, wherever the functions of a peripheral deviate from those described in 10.1 and 10.2 full details are given in the section on instructions for the relevant device.

The programmer will, of course, use whichever alternative is best suited to his type of program and his purposes; for example, normally in a background program he would use those functions which ensure time-sharing between independent programs. The actual values to be used in setting parameter F are given in the section for each device in Chapter 13 onwards.

10.1 Input Functions

In order to share calculation and input time within a single program, a buffer must first be 'prepared': the request to fill the buffer is attached to a queue (if any) for the specified device and control returns immediately to the main program. (Any number of 'prepare' orders can be given for the same device and they will all be queued in correct sequence.) The buffer is then filled automatically as soon as possible, whilst the main program is free to continue. When the buffer is actually required, a 'provide' instruction is given and, if the buffer is full, it is handed over to the calling program.

If the buffer is not yet full, the calling program waits for it; during this wait control passes to any other program which is ready to use the central processor. If parameter $B \neq 0$ in the 'provide' order, then the specified buffer is provided, but if B = 0, the next completed buffer on the queue is provided. This is the only case in which setting B = 0 does not require PCP to allocate a buffer area.

Alternatively, a 'prepare' instruction can be followed by an order to 'provide and prepare' buffers. In this case, the next available full buffer is handed to the calling program and another buffer as specified by parameter B (which may be zero) is prepared. The length of the queue for that device thus remains unchanged.

Generally a buffer is filled by a prepare instruction, a function which is time-shared with the calling program. However, this is not obligatory. If a provide order is given for a buffer not previously prepared, the request is queued and the calling program waits until the buffer is provided. If any other program can use the central processor during the wait control will pass to it. When input of the buffer is complete, the buffer is handed to the calling program which can then continue. If the queue is empty when a 'provide and prepare' order is given, two requests to prepare buffers are queued;, the first of these, always allocated by PCP, is filled while the calling program waits, and then provided. The second specified by parameter B (which may of course be zero) is prepared.

In both cases of time-sharing the 'provide and prepare' function is equivalent to a 'provide' with B = 0 (i.e. provide the next available buffer) followed by a'prepare'.

10.2 Output Functions

For all output functions parameter B of the macro 'buffer' must never be set to zero, but must specify the address of the buffer's codeword.

In respect of the time-scale and effect, output functions are broadly analogous to input functions. The equivalent of 'prepare' a buffer for input is 'output' a buffer. This causes a buffer to be put on the queue for output while control returns to the main program, which can then continue. Subsequently the programmer can given an instruction to 'wait' until the specified buffer has been output; in this way he can determine whether a buffer has yet been output and, if not, the calling program waits for it.

Alternatively, to ensure time-sharing between independent programs, the instruction to 'output and wait' should be given; the buffer is then placed on the appropriate queue, the calling program waits and control passes to any other program which can use the central processor. The calling program will not indicate to the control section of PCP that it is free to continue until the buffer has been emptied.

11. ERROR EXITS (EL)

11.1 Error Labels

All the macros available in PCP have as one of their parameters (EL) a label to which exit is made if the system detects an error, either in the programming or in the functioning of a device. This label (which specifies the entry point to another portion of program) must have been declared a global label as defined in SAP Mk. II; this means that is available to any part of the program within which it was declared.

11.2 Effects of an Error Exit

On detecting an error, the system tries where possible to correct this (e.g. by re-reading a pre-determined number of times from magnetic tape if a parity error occurs). If this is impossible, control returns to a specified point of the calling program. Although the buffer on which exit was made, and any other buffers on that queue, remain under control of the system, no further input or output takes place on the same device until, and unless, the calling program wishes to continue (see 11.4).

An error exit means that a jump is made to another part of the program; therefore there must be some way of indicating where to return after the error routine and of restoring the status quo: all the information required for the return is copied across from the continuation register, a part of the control list which keeps a "running commentary" on the state of the program, to 5 locations of a "SAVE" block for later use. This block is part of PCP's own workspace; its availability (i.e. whether it is already in use from a previous error exit) is determined by the setting of the ERREX marker. If the "SAVE" are occupied already, the program is held up until the ERREX marker indicates they are available. If this hold-up occurs, control may pass to any other program. This procedure is effected automatically by the control section of PCP, and unlike the interrupt scheme for the basic 503 system, does not entail any action on the programmer's part.

The portion of program to which control passes on exit, normally consists of a short diagnostic routine and, particularly in the case of machine errors, correction if possible.

11.3 Error Classes and Types

On exit to the error label, the class number of the error is held in the accumulator. Thus the first action of a diagnostic routine would be to determine the cause of the error from the number in the accumulator, as follows:

Class No.	Interpretation
1	Device Unavailable
2	Program error
3	Parity error
4	Special Magnetic Tape Errors.

In order to qualify the error further, each class is sub-divided into types. This information together with the device number is, where relevant, held in the auxiliary register in the following standard form:

00 Device Number : 00 Error Type No. *) P7

After the error class had been determined, the accumulator should be cleared. The device no. can then be brought to the accumulator by a 54 18 order; after clearing the accumulator again, a 54 20 order brings the error type no. to the accumulator. The table below shows the error types which exist for each class of error and for each device.

Where an error label has not been supplied in the parameter list, this is detected when the instruction is issued and

NO EL

is displayed before the program is removed from the PCP Control List.

Device	ERROR CLASS			
	1	2	3	4
	(Device Unavailable)	(Program Errors)	(Parity Error)	(Special Mag. Tape Errors)
Tape-readers Punches and Plotter *	-	See Standard Programming Error list below	-	-
Lineprinter	1. Manual 2. Paper nearly out or throat open	"	-	-
Card Reader	1. Manual 2. Read Error	"	-	-
Card Reader/ Punch	1. Manual 2. Read Error 3. Punch Error 4. Read/Punch Error 5. Prog. button set 6. End state (i.e. S1 and S2 empty	"	-	-
Magnetic Tapes	1. Manual	"	1. Read Error 2. Write Error 3. Wrong Parity specified	1.Tape Mark input 2.Tape Mark input and EOT passed 3.Past EOT on output 4.Writing not permitted 5.Function not allowed(B.O.T.) 6.Function not allowed(B.O.T.)

* It is not possible to give an indication by program when these devices are for some reason unavailable. The operator should, therefore, keep a close watch for any indication on the various lamps and buttons.

The conditions which cause a Class 2 (Standard Programming Errors) exit are defined under the following types:

Error Type	Interpretation
1	Device not attached to this program
2	Parameter B=0 for output instruction
3	Parameters D or N out of range
4	Parameter F out of range
5	Parameter M out of range.
6	Parameters C or K out of range.

Notes on Class 2 Error types:

1 Occurs when the macro

open device [D,N,EL] (see Ch. 7.3.)

has not been given for the required device, or when the device has already been attached to another program.

2 The specification of a buffer to be emptied (output) must always be given, otherwise the programmer would obviously have no control over the information to be output.

3 Parameter D must lie within the range

$0 \leq D \leq 7$. The range for N, where relevant, is defined in the section for each device.

4 & 5 The permitted ranges for parameters F, M, C, K are defined in the section for each device.

11.4 Error Messages

If the programmer wishes to have a message displayed indicating an error, he must make provision for this within the error routine. After analysing the information on type, class and device, an output order should be given to the typewriter to print out some form of message, for example:

LP, ERROR 1, TYPE 1
or LP IN MANUAL.

11.5 Continuation after Error Exit

The programmer, if and when he has dealt with the cause of error, has three courses of action available:

1. Return to the point in the program at which error exit occurred; this is held in one of the "SAVE" locations and results in the continuation registers being restored in accordance with the SAVE locations. The transfer of control is thus governed by PCP.

   This course is taken simply by writing the macro "return" at the end of the error routine.

2. Abandon the continuation point preserved in SAVE and re-enter the program at another restart point. This point is determined by the macro "restart" followed by parameters specifying the label and block name, thus:

   restart [label * block]

   The label used must be declared as global at the head of the program. (See SAP Mk.II, 2.4.1., Chapter 7.1).

3. Stop the program, i.e. return control to the supervisory program - STAR or RAP. This course, only taken when the error is irretrievable, is taken by writing the macro "stop".

12. OTHER PARAMETERS

12.1 Mode (M)

In general the mode specifies in what form information is to be input, stored and output. In binary mode, the form is normal 503 wholewords (see 503 Manual 2.2.1.); in character mode, one character is stored in bit positions 1 to 7 of each word, using the 503 standard 7-bit paper tape code.

Where the modes for a particular device vary from this, full details are given in the relevant section.

12.2 Special Purpose Parameters (C & K)

These are purpose parameters which take various meanings according to the needs peculiar to each device.

The most general use of C is to specify the size of a buffer area (for input) which is to be provided by PCP, i.e. when B = 0. The size specified must never exceed the maximum permitted size for a SPAN block i.e. 4096 locations. However, it can be omitted, in which case a buffer of standard size (121 words) is provided. For the card/reader punch and certain magnetic tape functions, C has special significance.

Parameter K is generally used as a terminating character, to delimit the data for input into a buffer.

---o0o---

The following chapters detail the actual values which parameters can take for each device. The general structure of the macro instruction "buffer" was given in Chapter 7, and explanation of the meaning and effect of parameters in Chapters 6-12.

13. TAPE READERS

The macro instruction is

```
                buffer [ D,N,B,F,EL,M,C,K ]
       where    D = 0  for tape readers
                N = 1 or 2 according to the No. of the reader
                B = 0  or address of buffer specification
                F = Function No. 1, 2 or 3
               EL = Label for error exit
                M = 0  for character mode, 1 for binary mode
                C = Buffer Size if B = 0
                K = the value of a 7 bit terminating character.
```

13.1 The Functions specified by Parameter F are:

F = 1 prepare to provide a buffer

F = 2 provide a completed buffer

F = 3 provide and prepare: provide a completed buffer and prepare another buffer

Briefly, 'provide and prepare' is equivalent to 'provide' with B = 0, followed by 'prepare'.

13.2 Error Exit (EL)

The conditions which cause an error exit are given in Chapter 11- 3 The permitted ranges for parameters when D = 0 (tape reader) are:

$1 \leq N \leq 2$

$1 \leq F \leq 3$

$0 \leq M \leq 1$

$0 \leq C \leq 4096$

$0 \leq K \leq 128$

13.3 Conditions for buffer full

(a) On character mode

Generally a buffer is considered full when one character has been input and stored in every location of the buffer area. However, if the character specified by parameter K or a Halt code(value 76) is input, the buffer is then considered full. The terminating character is stored in the current position of the buffer and the remaining words are cleared. This does not affect the size of the buffer as specified in the header-word.

If the option to complete a buffer by a terminating character is not required, parameter K can either be omitted (as a trailing null parameter) or set to 128.

(b) <u>On binary mode</u>

A buffer is normally considered full when six characters have been input and packed into each word of the buffer. In this mode parameter K has no significance. Since a Halt code cannot be recognised in binary mode, the buffer is considered full if four inches of blanks are read in (i.e. 40 blanks). The remaining words of the buffer area are cleared, but the size of the buffer as specified in the headerword is not changed.

13.4 End of Tape Messages

If a buffer is completed by reading a Halt character (in character mode)or by four inches of blanks (in binary mode), the message

```
EN TAPE   R N
```

is displayed, where N = 1 or 2 (no. of tape reader). Any further input instructions for that reader are delayed until a new tape has been loaded in the reader. The operator must depress the MESSAGE button and type

```
PCP;2;N.
```

where N = 1 or 2.

If a "provide" instruction is received before the new tape is loaded, the message

```
LOAD TAPE R N
```

will be displayed and the program is held up until the new tape has been loaded and the above procedure followed.

14. TAPE PUNCHES

The macro instruction is

```
buffer [ D,N,B,F,EL,M,O,O ]
```

where D = 1 for tape punches

N = 1 or 2 according to No. of tape punch

B = Function number 1, 2 or 3

EL = Label for error exit

M = 0 for character mode, 1 for binary.

14.1 Use of Character Printer

If the high speed character printer is connected to the system in the place of a tape punch, the macro instruction remains the same except that M must always be set to 0 (character), binary being invalid for printing.

14.2 Functions

The functions specified by parameter F are:

F = 1 output - place specified buffer on queue for output, return control immediately.

F = 2 wait - see if specified buffer output; if not do not return control until it is free.

F = 3 output and wait - place specified buffer on queue for output and do not return control until it is free.

14.3 Error Exit (EL)

The conditions which cause an error exit are given in Chapter 11. The permitted ranges for parameters when D = 1 are:

$1 \leq H \leq 2$

$1 \leq F \leq 3$

$0 \leq M \leq 1$

15. LINE PRINTER

The macro instruction is

buffer [D,O,B,F,EL,O,O,O]

where D = 2 for line printer
B = Buffer specification; buffer size = 121 words
F = Function 1, 2 or 3
EL = Label for error exit.

15.1 Functions

The functions specified by parameter F are:

F = 1 output - place specified buffer on queue output, return control immediately

F = 2 wait - see if specified buffer output; if not, do not return control until free.

F = 3 output and wait - place specified buffer on queue for output and do not return control until it is free.

15.2 Error Exit (EL)

The conditions which cause an error exit are given in Chapter 11. Note that error type 1 of Class 1 i.e. printer in Manual, may be the result of a variety of errors e.g. hammer drive fuse blown, paper runs away etc. Since these all cause the printer to go into the MANUAL state, they are grouped together.

The permitted range for F when D = 2 is

$$1 \leq F \leq 3.$$

16. ELLIOTT CARD READER

The macro instruction is

```
buffer [ D,0,B,F,EL,M,0,0 ]
```

where D = 3 for card reader
B = Buffer specification or zero; buffer size = 80 words
F = function no. 1, 2 or 3
EL = Label for error exit
M = 0 or 1 (explained below in 16.3)

16.1 Functions

The functions specified by parameter F are:

F = 1	prepare to provide a buffer
F = 2	provide a completed buffer
F = 3	provide and prepare provide a completed buffer and initiate reading of next buffer.

16.2 Error Exit (EL)

The conditions which cause error exit are given in Chapter 11. Full details of device error conditions which cause Class 1 exits, Manual and Read Error, can be found in 1.4.6 of the 503 Manual.

The permitted ranges for parameters where D = 3 are:

$$1 \leq F \leq 3$$
$$0 \leq M \leq 1$$

16.3 Modes

Parameter M does not in this case define how a buffer is to be coded. It is provided, to cater for a different card reader to be introduced as an optional model later in 1965. For the present 80 column Elliott card reader, M should be set to 0. Setting M to 1 will specify the optional card reader when it is introduced.

17. ELLIOTT CARD READER/PUNCH

The macro instruction is

buffer [D,O,B,F,EL,M,C,K]

where D = 4 specifying the card reader/punch
B = address of buffer's codeword; size of buffer = 80 words.

17.1 Functions specified by parameter F are:

F = 1 Read a Card. The first available buffer is taken from the read stack and read.

F = 2 Send Card. The card last read is sent to a given pocket (specified by parameter C).

F = 3 Punch a Card. Punch the contents of a specified buffer on a blank card and send it to a given pooket (specified by parameter C).

F = 4 Overpunch. Overpunch the card last read with the contents of the specified buffer (B) and send it to a given pocket (specified by parameter C).

N.B. All punching on cards from the main track, whether blank or not, is regarded as overpunching.

F = 5 Find Type. Find type number of next card on read stack. This only applies when reading ahead is allowed, which is governed by the value of parameter K, i.e. not allowed if K = 0, and allowed if K ≠ 0.

17.2 The Mode (M)

This is the actual operating mode of the machine (controlled by switches on the console) rather than an indication of how the buffer is to be de-coded. The values and the machine state they indicate as follows:

M = 2	Read only
M = 3	Punch only
M = 4	Read and Punch
M = 1	Read and Punch on both Main and Sub. tracks.

When overpunching, any attempt to punch into an existing hole is forbidden and may cause a parity failure. PCP will make no check to ensure that this does not occur.

17.3 Preselection of Pockets

The preselection of card pocket destinations is combined with output orders. The selected pocket is specified by parameter C as follows:

C = 1	for pocket 1
C = 2	for pocket 2
C = 3	for main pocket

The preselect order always over-rides any previous pocket selection for the card in question. If, however, a card is subsequently punched and found to be in error it will automatically be sent to pocket 1.

If no preselect order is given, cards originating from the main hopper will go to the main pocket and cards from the subsidiary hopper to pocket 2. In each case punch errors cause the error cards to be sent to pocket 1.

17.4 Error Exits

The conditions which cause error exits are given in Chapter 11. Further details of the six types of Class 1 errors can be found in the functional specification for the card reader/punch (1.4.4 of the 503 Manual).

If a read error (type 2) occurs, the operator should reload the cards present on the main track using the ejection and reloading facilities provided on the device. If a punch error (type 3) occurs, the buffer of information punched onto that card is retained in order that the instruction can be repeated to punch the buffer onto another card. However, in the case of punching onto a non-blank card, immediate repunching is not possible.

The permitted ranges for parameters when D = 4 are:

$$1 \leq F \leq 5$$
$$1 \leq M \leq 4$$
$$1 \leq C \leq 3$$
$$0 \leq K \leq 128$$

18. DIGITAL PLOTTER

The macro instruction is

```
buffer [ D,O,B,F,EL,M,O,O ]
```

where D = 5 for digital plotter
B = address of buffer's codeword
F = function, 1, 2 or 3
EL = label to which error exit occurs
M = 1 for unpacked mode, 2 for packed mode.

18.1 Functions

F = 1	output	- place specified buffer on queue for output, return control immediately.
F = 2	wait	- see if specified buffer output; if not, do not return control until it is free.
F = 3	output and wait	- place specified buffer on queue for output and do not return control until it is free.

18.2 Error Exits

The conditions which cause error exits are given in Chapter 11. It is not possible to give an indication by program when the plotter is for some reason unavailable, (i.e. Error Class 1 does not exist). The operator should, therefore, ensure that the plotter is set up correctly for operation and keep a careful watch on the switches, paper, etc.

The permitted ranges of parameters when D = 5 are:

$$1 \leqslant F \leqslant 3$$
$$1 \leqslant M \leqslant 2$$

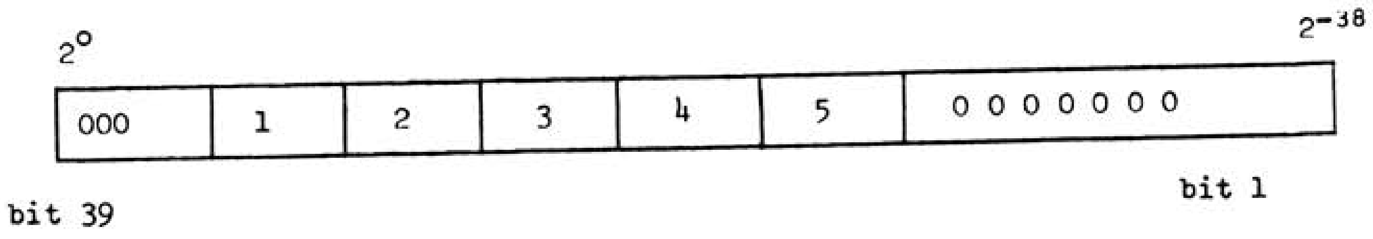
18.3 Modes

Both modes provided deal with 6 bit characters; the significance of these is described in 1.4.7 of the 503 Manual. The character is added to a 72 7168 instruction to form the actual instruction obeyed.

If M = 1 there is one character in each word of the buffer (unpacked).

If M = 2 there are six characters packed into each word of the buffer. The top three bits of each word are zero (i.e. the 3 most significant bits). The most significant character is the first to be used.

If there are less than six characters in a word, the word must be left justified, i.e. any unused space is at the least significant (right) end of the word.



For example, the above word contains 5 characters, numbered in the order in which they are used.

19. TYPEWRITER

The macro instruction is

buffer [D,O,B,F,EL,O,C,K]

where	D = 6 for the typewriter
	B = Buffer specification
	F = function 1 or 2
	EL = label to which error exit is made
	C = buffer size(if B = 0)
	K = a 7-bit terminating character.

19.1 Functions

Since messages via the typewriter usually require immediate action, any instruction for the typewriter has absolute priority; it is obeyed immediately whilst the program waits, so that there is never a queue of instructions waiting for the typewriter.

F = 1 input	- the program waits until the message/data has been typed and input into the buffer. The parameters B,C and K have the same significance as for tape reader input.
F = 2 output	- the program waits until the buffer has been printed out. This function might occur particularly in error routines to print out messages in accordance with the program's analysis of class and type.

20. MAGNETIC TAPE HANDLERS

The 503 magnetic tape systems functions under the scheme of variable length blocks. The length of a block is determined by one program write instructio and may be of any length between 3 and 4096 computer words.

On output a buffer of information in store is written to tape as one block. On input one block is read into a buffer: if the block is longer than the buffer, the remaining words on tape are skipped over and not stored, if shorter, the end words of the buffer are cleared.

Information is recorded on tape usually either in binary or character form with odd parity. However, an even parity character mode is available for the sake of IBM compatibility. In the odd parity modes a block numbering system is used: on output each block is given a number which is used to identify it in"tape move"and input instructions thereafter.

The system automatically carries out certain checks and gives indications on detecting any errors.

The rest of this chapter deals fully with magnetic tape in relation to the PCP magnetic tape functions. However, for any reader unfamiliar with magnetic tape systems in general, it is advisable to read section 1.4.2 in Vol.1 of the 503 Manual in conjunction with this chapter.

20.1 Macro Instructions

Two macro instructions are provided to cover the range of functions for magnetic tape. Firstly, the normal input/output macro:

```
          buffer   [D,N,B,F,EL,M,C,K]

          where    D = 7 for magnetic tape handlers.
                   N = handler number, 1 ≤ N ≤ 8
                   B = Buffer specification
                   F = function number 1 to 9 or 14.
                  EL = label to which error exit occurs
                   M = recording mode, 1, 2 or 3.
                   C = size of buffer if B = 0
                   K = number of block for input or output
```

Secondly, a separate macro used only for moving backwards and forwards along tape, and not involving any transfer of data:

tape move [N,F,EL,C]

where N = handler number, $1 \leqslant N \leqslant 8$

F = function number, $10 \leqslant F \leqslant 13$

EL = label to which error exit occurs

C = number of blocks to be traversed

20.2 Recording Modes

There are three ways of recording a 39-bit computer word of magnetic tape. This is determined by setting parameter M as below. A block must be input using the same mode in which it was output.

M = 1 Binary mode with odd parity

All 39 bits of each word are recorded with odd parity in seven 6-bit sets, the top three digits of the first set being zero.

M = 2 Character mode with even parity

The 3 most significant bits of each word are ignored; the remaining 36 bits are recorded with even parity in six 6-bit sets. On re-input the 3 most significant bits of each word stored are zero. This mode is compatible with IBM binary coded decimal tapes.

M = 3 Character mode with odd parity

This mode is identical to M = 2 except that odd parity is used.

20.3 Input

20.3.1 Input Functions

The input functions specified by parameter F are:

F = 1 prepare to provide a buffer; the request is queued and control returns to the main program.

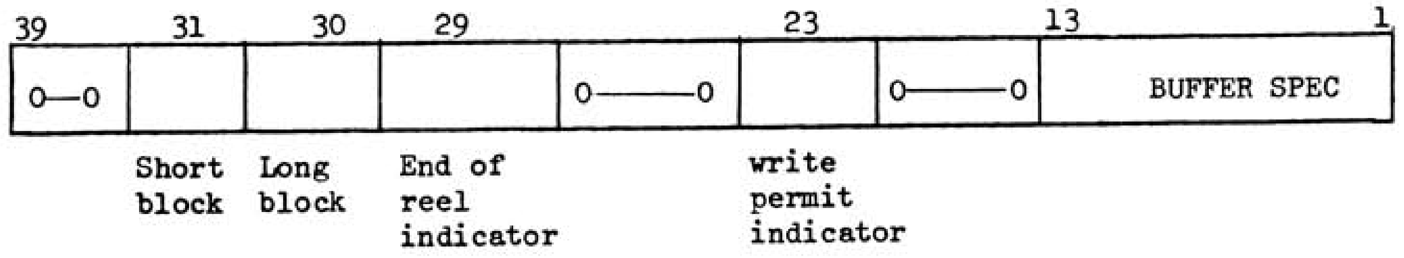
F = 2 provide a completed buffer

F = 3 provide and prepare - provide a completed buffer and prepare another buffer

F = 4 provide immediately:- this is a special function used to input a block which has been written beyond the end of reel marker. The instruction is placed at the head of the queue and carried out immediately.

20.3.2 Control Word

When control returns to the main program after an instruction with F = 2, 3 or 4, the accumulator holds a control word giving information on the buffer provided, as follows:



(i) The buffer specification is (as normal) in the 13 least significant digits.

(ii) Writing is permitted on tape (i.e. the write permit ring has been inserted) if bit 23 is set to 1. If it is zero, and the programmer wishes to write on tape, he should arrange for a message to be output for the operator's attention.

(iii) If bit 29 is set to 1, the tape is at the end of reel marker. To input a further block, the special function F = 4 must be used.

(iv) If bits 30 (31) are set to one, then the block on tape was longer (shorter) than the size of the buffer.

The number of the block just input is found in the first word of the buffer (for modes 1 and 3 only).

20.3.3 Special Purpose Parameter K

This parameter is only significant for modes 1 and 3. If K = 0, then the next block on tape is input. If K ≠ 0, the block whose number is specified by K is input. Any advancing or retreating over tape in order to arrive at the specified block is carried out automatically.

20.4 Output

20.4.1 Output Functions

The output functions specified by parameter F are:

F = 5 output - place specified buffer on queue for output, return control immediately.

F = 6 wait - see if specified buffer output; if not, do not return control until it is free.

F = 7 output and wait - place specified buffer on queue for output and do not return control until it is free.

F = 8 output tape mark - write special single character block used to delimit files and reels etc. (see section 8); this must be written in mode 2, i.e. even parity character mode. Parameter B should be set to zero; this will not cause an error exit, as happens in all other output instructions.

F = 9 output and wait immediately - this instruction is the output equivalent of "provide immediately" with F = 2. It is placed at the head of the output queue to output data beyond the end of reel marker.

F = 14 Output tape mark immediately - this instruction is provided so that the user may, if he wishes, adopt the standard procedure of outputting a tape mark after the end of reel marker; the order is placed at the head of the queue and the mark output immediately.

20.4.2 Block number parameter K

This parameter is only significant for modes 1 and 3.

If K = 0 the block is written on tape at its current position. If K = 1 the block is written after the highest numbered block written since the tape was currently opened, i.e. any writing on tape from a previous run is ignored.

If the tape is only in use for output both values of K have the same effect. But if after output of several blocks it is required to retreat and re-read some of them and then continue writing where one left off before, then parameter K = 1 is used. In this case any advancing or retreating required to arrive at the specified block or position is carried out automatically, and a check made that the tape halts at the correct block number.

However, if the output is in fact selective overwriting, the programmer must deal with tape movement and checks.

20.5 Tape Movement

Tape movement is, in effect, an off-line operation, so that as many of the handlers as required can carry out one of these functions simultaneously.

For the "tape move" macro, the number of blocks to be moved as specified in parameter C. The functions specified by parameter F are:

F = 10	advance	- C blocks, where C>0. This causes C blocks to be skipped.
F = 11	retreat	- C blocks, where C>0.
F = 12	rewind	- positions the tape at the beginning of reel marker. If the tape is already in that position, the instruction is ignored.
F = 13	erase	- the next four inches of tape are erased.

20.6 Block Numbering System

This sytem applies to modes 1 and 3 (odd parity modes) but is not used for even parity character mode.

Each block written is numbered sequentially by the system and its number stored in the first word of the block; on input this number is checked to ensure that the required buffer is being input.

20.6.1 Block Counts

Three counts are kept for each handler within PCP's workspace. These are available for use by any of the other systems, e.g. SPAN, STAR, SAP2. They may be referred to as detailed in the three sections below provided the following replacements are declared at the head of the program:

current block [;HM], last block [;186], block max [;194]

(i) current block count holds the number of the block which has just been written, read or advanced over or one less than the block just retreated over. Before an output instruction is obeyed, the current block number is stored in word 1 of the buffer in store, so that it will be output to tape in the correct position.

On input the block number in word 1 is checked against the current block count. If they are not equal, the system initiates a search until the required block is reached. This may occur as a result of 'noise' blocks due to irregularities in the tape surface: if one of these spurious blocks is encountered it is considered as a legitimate block written by program and will throw the counting system out of step. Thus the necessityfor a checking system.

In fact, due to the possibility of these noise blocks,after any retreats or advances the tape position is considered unchecked. The current block count has a marker to indicate that it is unchecked. If an input instruction then occurs, the unchecked state is cancelled, since the input instruction itself incorporates a check. Output instructions will not be obeyed if the tape is in an unchecked state, but see section 2 of this chapter.

(ii) last block on queue holds the number of the block referred to in the last instruction on the queue (or one less for a retreat instruction). If the last instruction was to advance or retreat, there is a marker indicating that the last block on queue is unchecked. On return to the main program after an output instruction has been placed on the queue, the number of the last block on the queue is also in the accumulator (in bits 1-16 inclusive).

(iii) block max. holds the number of the highest numbered block put on the queue, and also the mode of this block:

39-37		16 - 1	
mode 1 or 3	0 0	block no.	location block max

This count is used to determine the position on tape for output if K = 1. The last section of a tape may effectively be deleted if block max and the mode are reset to the number and mode of the last block in which the programmer is interested.

20.6.2 Input and Output on same tape

After advancing or retreating, the position on tape is unchecked, as stated in 20.6.1(i). It is not, therefore, safe for an output instruction to be executed immediately, since this may result in overwriting wanted blocks. The position on tape must first be checked by an input instruction; this is done automatically when the instruction is to output after the highest numbered block written so far, i.e. output instruction with K = 1. But if the programmer wishes to overwrite blocks already written (output with K = 0) he should check the position himself by giving an input instruction.

If any tape movement is to occur between writing blocks, the instruction to write the first block must be followed by an erase instruction (or output of a dummy block e.g. output tape mark). This is because after reading a block and then returning to write the second block, the tape may not stop in exactly the same place as it did after writing the first block and thus spurious digits may be left between the two blocks. Therefore, a retreat after output of a block whose number is equal to block max is preceded by an erase instruction. This is carried out automatically. However, if the blocks for output entail selective overwriting, the programmer himself should give the erase instruction.

Note especially that if the programmer wishes to overwrite a tape written during a previous run, he must set the count in block max equal to the last block on tape in which he is interested. Otherwise, under the scheme outlined in the preceding paragraph,wanted blocks may be accidentally erased.

SUMMARY

In short, when a tape is used solely for input, solely for output, or for both but with no selective overwriting, all checks are carried out automatically by the system. However, all safety precautions for overwriting are the programmer's responsibility. As a general rule a wide safety margin of blank tape should be allowed both after the block to be overwritten and after the replacement block. Overwriting is best carried out with full understanding of the magnetic tape system rather than by mechanical rule; full details can be found in 1.4.2 of the 503 Manual, particularly in the section Special Effects.

20.7 Tape Markings

20.7.1 Tape Mark

This is a special single character block written on even parity (character mode), used if required to delimit files of data on tape and always to indicate the end of writing on tape. A tape mark is output using the special function "output tape mark" (F = 8); it is the only exception to the rule for minimum block size of 3 words.

Whenever a tape mark is input, an alarm exit occurs to the error label (EL), giving an error class 4, type 1 indication (see Chapter 21). This type of indication is given for every tape mark except the last, (for which, see section 3 of this chapter). This in no way implies an error but is merely so that the programmer knows the tape is positioned at the end of a file of data.

20.7.2 Beginning of Tape

On each reel of tape there is a physical beginning of tape marker (BOT) just before the position where the first block can be written on tape. A tape is positioned at this marker when it is loaded manually or when a rewind instruction (F = 12) is obeyed. When the tape is already in this position , a rewind instruction is ignored and a retreat instruction causes an error exit (error class 4, type 5).

20.7.3 End of Tape

There is a similar physical end of tape marker (EOT) about 25 feet before the end of a reel of tape. When on output the system encounters this marker, an alarm exit occurs indicating that the handler is in the end of reel state (error class 4, type 3). If he is adopting the standard practice, the programmer should then output a tape mark using the function'output tape mark immediately'(F = 14); thereafter he may write further blocks using the function 'output and wait immediately' (F = 9). There is no actual program restriction on the number of blocks which may be written beyond the EOT marker, so the programmer must ensure that the safety limit of approximately 15 feet beyond the marker is not exceeded: it is recommended that not more than 2 blocks of 256 words be written.

On input the alarm exit occurs when both the EOT marker has been passed and the tape mark which follows it has been input (error class 4, type 2). This indicates that any further blocks written beyond this point can only be input using the function 'provide immediately' (F = 4).

Rewind and retreat instructions are also allowed when the handler is in the EOT state, but all other instructions cause an error exit (class 4, type 6).

20.8 ERRORS

The conditions which cause error exits are given in the table in Chapter 11. The error classes and types are rather more complex for magnetic tape handlers, so further amplification where necessary is given below.

20.8.1 Programming Errors (Class 2)

The permitted ranges of parameters when D = 7 are:

$1 \leq N \leq 8$

$1 \leq F \leq 14$

$1 \leq M \leq 3$

buffer size $3 < C \leq 4096$, or $C = 0$

tape move $C > 0$

20.8.2 Parity Errors - Class 3

Within this class three types of indications are given (as shown in the table in Chapter 11) so that the programmer can determine which form of parity error has occurred - read error, write error or wrong parity specified. All checking involved in the detection of parity errors is carried out automatically by the system; for interested readers a fuller description of how this is done is given below.

Type 1 - Read Error and Type 3 - Wrong Parity

If a parity error occurs on reading a block and if there is also a short block indication set, this may be either noise on tape or a tape mark read on wrong (odd) parity. The block is first re-read by an instruction set to mode 2 (even parity, character - same mode as a tape mark) and with block size set to 3 words. If on re-read there is still a short block indication set, then it must be either noise or a tape mark (since it is <3 words long). The system then tests the short block to see if it is a tape mark: if it is, an alarm exit occurs indicating a tape mark input (see 20.7.1). If it is not, then it must be noise on tape; in this case, the pseudo-block is ignored and the original instruction is repeated to read the next block.

If the short block indication is not set, either in the first instance or on the special short re-read (see Type 1 above) and if the mode used is 2 or 3 (character), it is re-read on the opposite parity to that originally used. If there is then no parity error, an error exit (type 3) occurs indicating that wrong parity has been specified in the instruction.

However, if on re-reading there is still a parity error, then up to four further attempts are made to re-read the block using the original parity specified. If any of these is successful, the program continues; otherwise after the fourth attempt an error exit occurs (type 1).

Type 2 - Write Error

Each block written on tape is checked immediately. If a parity error has occurred, an attempt is made to rewrite the block. If there is now no parity error the program continues; otherwise the cycle: retreat 1 block, erase, rewrite, is repeated. If there is still a parity error after a total of four attempts to rewrite, an error exit occurs (type 2).

If the EOT marker has been passed when a parity error is found, only one attempt is made to rewrite the block.

20.8.3 Special Magnetic Tape Errors - Class 4

This class of errors refers only to magnetic tape and includes exits which are not strictly errors, but rather an alarm indication of the tape's position.

Types 1, 2 and 3

These three types are alarms to warn the programmer of specific conditions so that he may take appropriate action. They indicate on input either a tape mark input (1) or the EOT passed and a tape mark input (2), on output that the EOT is passed and therefore very little tape remains.

Type 4

This error exit occurs if the programmer attempts to write on a tape which does not carry a 'write permit' ring. It is in addition to the indication given by bit 29 of the control word on exit from a provide instruction. In fact, this error should only occur once in a program: the first time it occurs a message should be output requesting the operator to insert the ring.

Types 5 and 6

Apart from the normal operating state of a tape handler, the programmer has indication of two other states, the BOT and EOT states; these must be treated in a special way as described in the previous chapter. The following table shows how each of the fourteen instructions are treated when a handler is in either of the states; 'error' indicates that the program will jump to the label specified by parameter EL when the instruction comes to be obeyed.

INSTRN		STATES	
		B.O.T	E.O.T
1	Prepare		Error
2	Provide		Error
3	Prepare and Provide		Error
4	Provide Immediately		
5	Output		Error
6	Wait		
7	Output and Wait		Error
8	Output TM		Error
9	Output and Wait Immediately		
10	Advance		Error
11	Retreat	Error	
12	Rewind	Ignored	
13	Erase		Error
14	Output TM Immediately		

Device	Chapter Ref.	D Device Class number	N Device number	B Buffer Spec.	F Function	EL Error Label	M Mode	C Special Purpose	K Special Purpose
Tape Reader	13	0	1 or 2	Address of buffer's codeword or zero	2. Provide 1. Prepare 3. Provide & prepare	Exit made for program errors (Class 2)	0 Char. 1 Binary	Buffer size if B = 0. If C = 0 standard size required	Terminating Character
Tape Punch	14	1	1 or 2	Address of buffer's codeword	1. Output 2. Wait 3. Output & Wait	Exit made for program errors (Class 2)	0 Char. 1 Binary	-	-
Line Printer	15	2	-	Address of codeword of buffer, 121 words in size	1. Output 2. Wait 3. Output & Wait	Exit made if device unavailable (Class 1) or program errors (Class 2)	-	-	-
Elliott Card Reader	16	3	-	Address of codeword of buffer, 80 words in size, or zero	1. Prepare 2. Provide 3. Provide & Prepare	Exit made if device unavailable (Class 1) or program errors (Class 2)	-	-	-
Card Reader/ Punch	17	4	-	Address of codeword of buffer, 80 words in size	1. Read a card 2. Send card to special pocket 3. Punch card & send to special pocket 4. Overpunch & send to given pocket 5. Find type of next card	Exit made if device unavailable (Class 1) or program errors (Class 2)	M specifies the operating mode of the device 1 = Read only 2 = Punch only 4 = Read & Punch 4 = Read & Punch on both tracks	Pocket No. for functions 2,3 and 4. 1 = Pocket 1 2 = Pocket 2 3 = Main pocket	Reading ahead for function 5 allowed if K ≠ 0, not allowed if K = 0
Digital Plotter	18	5	-	Address of codeword of buffer	1. Output 2. Wait 3. Output & Wait	Exit made for program errors (Class 2)	1. Unpacked chars. i.e. one in each word 2. Packed chars. i.e. 6 in each word	-	-
Typewriter	19	6	-	Address of buffer's codeword, or zero	1. Provide 2. Output	-	-	Buffer size if B = 0. If C = 0 standard size requ.	Terminating Character
Magnetic Tape Handlers "buffer macro"	20	7	Handler No. 1 to 8	Address of buffer's codeword, or zero	1. Prepare 2. Provide 3. Prepare & Provide 4. Provide immediately 5. Output 6. Wait 7. Output & Wait 8. Output TM 9. Output & Wait immediately 14. Output TM immediately	Exit made if device unavailable (Class 1), program errors (Class 2), parity errors (Class 3), special mag. tape errors (Class 4).	1. Binary, odd parity 2. char. even parity 3. char. odd parity	Buffer size if B = 0 If C = 0, standard size req.	<u>INPUT</u> K = 0, input next block K ≠ 0, input specified block K. <u>OUTPUT</u> K = 0, output at current position of tape K = 1, output after highest numbered block written.
"tape move" macro	20	-	Handler No. 1 to 8	-	10. Advance 11. Retreat 12. Rewind 13. Erase	-	-	Number of blocks to be traversed for functions 10 and 11.	-

APPENDIX 2

Program Running Procedure

Programs are now run under the combined control of PCP and RAP. Programs which make use of PCP must be triggered via PCP in the manner described below. The first entry to PCP enables the operator to then type exactly the same message as he would were the program a SAP1 program and the message rejection works in exactly the same way (except that in certain cases, when a mistake is made, only the current identifier need be repeated rather than the complete message).

The user may either run one (main) program alone or he may choose to run a background program (see 2.5.3.1 definition) in conjunction with the main program. The sample messages below show how this may be done.

To run the main program, alone, type:

```
PCP."PROGRAM NAME";"TRIGGER NUMBER".
```

To run both main and background programs, type:

```
PCP."PROG1 NAME";"TRIG1";"PROG2 NAME";"TRIG2".
```

APPENDIX 3

PCPP

INTRODUCTION

PCPP is a version of PCP which is coded by SAP Mark 1 and is independent of SPAN. Administration of the free main store available for buffer transfers is performed by PCPP (see SPECIAL FEATURES, No.2).

PCPP provides time-sharing facilities on machines with or without core backing store. If it is used on a 503 which has core backing store fitted, the programmer may make use of the backing store provided he follows the precautions detailed later (SPECIAL FEATURES,No.4).

In general, PCPP performs the same functions as PCP, although certain restrictions are imposed with regard to the facilities provided and these are also changes in the method of use. These are detailed below.

References in round brackets (where given) are to the appropriate section of the description of PCP, where greater detail may be found.

TAPE

Binary code for input by RAP.

STORE USED

Approximately 1800 locations, including workspace.

RESTRICTIONS

1. The device booking facilities are not available, i.e. all devices are available to each program. The user must therefore ensure that any two programs being run together do not interact, and that output queues are empty before the program ends - (2.5.3:7.3).

2. If two programs are run together, only one may obtain its buffers through PCPP by use of the zero setting for parameter B.

3. (2.5.3:7) Replacements for the PCPP macros are not included as standard in SAP1, and the following replacement statements must be declared at the head of the program if they are called:-

```
buffer [ D,N,B,F,EL,M,C,K;
         72 0 : 30 F
         COMP, PCPP, 2
         00 K : 00 B
         00 D : 00 N
         00 C : 00 M
         40 EL: 00 0 ]
```

```
BSUPPLY [ SIZE; 72 0 : 30 SIZE
                COMP, PCPP, 6 ]


BRETURN [ BSPEC; 72 0 : 30 BSPEC
                COMP, PCPP, 7 ]
```

4. The 'tape move' macro may be written as a special case of the 'buffer' macro by using the appropriate function value (2.5.3:20.1)

5. All parameters must be supplied, though those not required by PCPP will be ignored. Trailing null parameters may not be omitted.

SPECIAL FEATURES OF PCPP

1. (2.5.3:8) The devices controlled by PCPP are:-

   Magnetic Tapes

   Line Printer

   Tape Readers

   Tape Punches

   Typewriter

2. PCPP assumes that the first and last free pointers of RAP indicate the free store from which it may allocate buffers at the request of the user's program.

If PCPP is unable to supply the space requested through the BSUPPLY macro, the accumulator will be zero on return to the user's program. (2.5.3:4 and 2.5.3:9).

3. In order to safeguard the error retrieval mechanism, PCPP writes a block number in bits 21-36 (inclusive) of the first word of each block written on magnetic tape. No significance is attached to the value of parameter K in the 'buffer' instructions.

4. Use of core backing store

PCPP provides no control of core backing store but interrupts caused by ADT transfers between main and backing store will nevertheless send control to PCPP. If this happens shortly after a '76' (prepare transfer) instruction for backing store has been issued by the user's program, then the effect of that instruction may well be destroyed by group 7 instructions issued by PCPP.

To avoid the above effect, the programmer using core backing store must prevent the occurrence of interrupts between the execution of the 76 and 77 instructions by the following method:-

```
        .
        .
        .
  72    0      ┐
        .      │
  76    BS     │
        .      │
        .      ├  No Interrupts Allowed
  77    X      │
  67    7886   ┘
  72    256
        .
        .
        .
```

5. <u>Error Indications</u>

The following list gives the standard error (STERR) indications which may be displayed, together with the definition of the error.

PROGRAM ERRORS

INDICATION		DEFINITION
STERR 1		No parameter specified where one is required.
STERR 2		Parameters N,F or M out of range
STERR 3		Parameters C or K negative, or D out of range
STERR 7		Attempting to remove a request from a non-existent queue: this is a PCPP error and should be reported if it occurs.
STERR 22	Magnetic Tape Errors	Instructions not allowed (see 2.5.3:20.8.3)
STERR 23		Buffer size error (e.g. less than 3 words)
STERR 24		Parameter B error
STERR 25		Handler number error
STERR 26		Function value error

RUN-TIME ERRORS

MESSAGE	ACTION
CANNOT READ BLOCK "B" ON H"N" CTLWD "INTEGER"	Type CONT. to cause the block to be ignored, or type STOP. to end off. The value of the last handler control word is displayed to assist the decision.
CANNOT WRITE BLOCK "B" ON H"N" CTLWD "INTEGER"	As above; continuation is not normally worthwile.
WPERR HANDLER "N"	Writing is not permitted on handler N. Insert the write-permit ring, and the program will continue as soon as the handler is reloaded.
H"N" NOT READY	Handler N in manual. Set handler to remote and change keyboard sign to continue.
LP NOT READY	Lineprinter is in manual. Set it to auto and change keyboard sign to continue.

The Controlling Programs

Programs to be run under the control of the Mark 2 system, fall into the following two categories: those which make use of the Peripheral Control Program (PCP) and those which do not. Just as under the Mark 1 system, overall control of the machine is through the Reserved Area Program. Programs not making use of PCP for peripheral activity may be triggered through RAP exactly as under the Mark 1 system. Programs which use PCP are run under the control of PCP and the method of program entry is amended accordingly. Once the operator has entered PCP, the message to cause entry to a program is the same as if RAP were receiving it. Precise details of the form of the message will be found in the Appendix to the PCP description.

When programs are to be run under the control of STAR, then RAP must be replaced by a special STAR version of RAP known as "STRAP". Certain of the facilities of RAP which, under the Mark 2 system, have no useful application, have been replaced by such facilities as are described in detail in section 2.5.5.10. Broadly these enable STRAP to detect whether or not STAR is in operation. STRAP may also be used to replace RAP for normal operation except for the use of the facilities which have been removed.

When a program ends and returns control to STRAP by means of the STOP instruction, a test is made to determine whether STAR is controlling the running of programs. If so, then STRAP sends control to STAR for the next command to be obeyed. In this case 'END' is not displayed on the output writer. When STAR is in control, all programs being run are run under the immediate control of PCP whether or not they make use of PCP.

When control comes to STRAP as a result of an error interrupt or manual interrupt then the operator has the option of making a decision as to whether to make a diagnosis (in the case of error interrupt) before sending control back to STAR. The precise details on which such control decisions are based will be found in the description of STRAP.

503 TECHNICAL MANUAL

VOLUME 2: PROGRAMMING INFORMATION

PART 5: PROGRAMMING AND OPERATING AIDS FOR THE NON-BASIC 503

SECTION 5: SEGMENTED TAPE ADMINISTRATIVE ROUTINES (STAR)

CONTENTS LIST

PREFACE

STAR is an operating system which improves the operating efficiency of the 503 by allowing the user to hold programs and compilers on magnetic tape. In this way the amount of paper tape handling necessary is minimised and throughput is increased. Once a program is on magnetic tape, the user may edit it before the next run by supplying a set of edit instructions on paper tape.

The operating instructions for each STAR run, during which many jobs will be processed, are prepared before the run and the complete set is read in and obeyed in sequence.

The STAR system is most suitable for the updating and running of programs under test. It is not suitable for running a sequence of production jobs since it is not possible to omit the compilation stage and no facilities are provided for producing binary versions of the programs on magnetic tape.

STAR

1. INTRODUCTION

1.1 The Purpose of STAR

The STAR system aims to make easier the updating and running of programs on the 503. It allows the programmer to store on a segment of magnetic tape all the programs and data he is currently developing, and it allows the contents of a batch of such segments to be modified and later used one after the other with the minimum of clerical work and operator intervention. The complete set of instructions for the processing of the batch is read into the machine before the processing begins, and thereafter the operator has only to keep the devices loaded and to obey any instructions that STAR may give via the output writer.

STAR operates on a non-basic 503 with a minimum configuration of a lineprinter, 3 magnetic tape handlers and backing store. The peripherals are administered by SPAN and PCP, with the exception of the tape readers and punches.

Many of the ideas used in STAR are similar to those of the Programmer's Utility Filing System (P.U.F.S) developed by Dr. M.V. Wilkes of Cambridge University. Other ideas have been borrowed from the very powerful Compatible Time-Sharing System (C.T.S.S.) developed for the 7090 at Massachusetts Institute of Technology.

1.2 The STAR Run

A STAR run is divided into 2 phases. During the first phase a new STAR batch tape is created containing mnemonic programs and their data; these are either new versions input from paper tape, copies or edited versions of existing programs on the old batch tape. Programs specified by the user are translated and run during the second phase. These programs must all be written in the same language although programs in various languages may be stored in the same segment/batch. The running program may use either one set of data from the new batch tape or from an external source, or both or neither. e.g. An external source is any device other than handlers 1,2.

1.3 EXAMPLE 1.

Suppose that the programmer is developing 2 programs, one written in SAC and the other in FORTRAN say, and he wishes to make use of the STAR system. Then he should prepare a list of commands to input, translate and run his programs as follows:-

Command	Purpose of Meaning
batch;0,FORT.	This usually identifies the old batch tape. In the case of the first STAR run there is no old batch tape. FORT is the name of the batch to be used in phase 2.

Command	Purpose or Meaning
new;seg1.	This opens a new segment called seg1.
inputc;testS.	This inputs the SAC program, testS, and writes it onto the new batch tape.
inputc;dataS.	This inputs the data for testS.
new;seg2.	This opens a new segment, called seg2.
inputc;testA.	This inputs the FORTRAN program testA.
inputc;dataA.	This inputs data for testA.
FORTRAN;3;testA;dataA.	The program testA is translated by FORTRAN (the 3rd entry to FORTRAN is a special one used by STAR) and run using data (dataA) from the new batch tape.
]	End of the commands tape.

To run the SAC program, the following commands tape is needed:

Command	Purpose or Meaning
batch;1,SAPBATCH.	Load what was the 'new' batch tape on handler 3. SAPBATCH is the batch to be used in phase 2.
update;seg1.	Prepare to process seg1.
SAP2;4;testS,10,20;1;dataS.]	The program testS is assembled by SAP2 (entry 4 is a special one used by STAR) and run using data (dataS) from the new batch tape. 10,20 are the estimates required by SAP2.

N.B. seg2 is copied across without alteration.

2. The STAR Tape

2.1 Segments and Files

A reel of magnetic tape is used to hold one batch of programs, each program comprising one STAR file. The batch is composed of a number of segments, each of which is processed independently of the rest of the batch. All files specified by the user in one segment are edited/copied/deleted/output (phase 1) or translated and run (phase 2) before STAR processes the next segment. It follows that data files must be in the same segment as the program files with which they are associated. Each segment is divided into files as specified by the user. The file is the basic unit on which STAR operations are carried out.

A file is written on the magnetic tape as a sequence of SPAN blocks of 122 words each, including the headerword. All files are written in binary mode with 5 7-bit characters per word.

Odd parity is always used on magnetic tape.

The last block in each segment is the segment directory. This contains an entry for each file in the segment. A file is named by an identifier of up to 6 alphanumeric characters, together with an integer version number. This allows the user to have several versions of the same file, each being the result of modifying the previous version. When a file contains a program its name should be that of the program. Each directory is preceded by a 5 word dummy block in binary mode containing <-1> in its 2nd, 3rd, 4th and 5th words. The last segment directory is followed by a 5 word binary block with <9LAST> in 2nd word.

If the user calls for a file by its identifier, he accesses the latest version; if he refers to <identifier>/n, he accesses the last version but n. Version numbers are updated automatically by STAR. Up to 8 versions of a file may be held in the segment i.e. versions 0 (the latest one), 1, 2, ... 7. All files with version numbers greater than 7 are deleted automatically by STAR.

2.2 Layout of the Tape

The first block on the batch tape after the reel label contains the batch directory, which has an entry for each segment. The batch directory is followed by several erases, a 5 word dummy block, and the individual segments.

3. Operation of a STAR Run

3.1 General

A STAR systems tape must be prepared by the user before the first STAR run. This contains all the programs required during phase 1 of the STAR run and is loaded on handler 1 (see 'Operating Instructions' for details). Handler 3 holds the old batch tape. Handler 2 holds the new batch tape and has writing permitted. Handlers 1 and 3 must be set to read only. No other program may write on these handlers during the STAR run.

One compiler batch must be prepared (see 9.3) also, for each language used by the programmer, in addition a further batch is required when running ALGOL programs. The two batches required for ALGOL must be on the same magnetic tape. The SAP2 and FORTRAN batches may be written on any magnetic tape, and in fact, all the batches required for phase 1 and phase 2 may be put on the same magnetic tape, although this increases the loading time and will reduce the speed of phase 2, which is not advisable.

The first step in the operation is to input RAP MT under the initial instructions and then to protect the reserved area.

The operator must type IN; 'name of phase 1 batch'. to load the STAR batch from Handler 1 into main and backing store, and transfer control to STAR. While this is going on, the operator should load the commands tape in TR1.

When the store is set up, STAR inputs the commands from TR1 and obeys them one by one. All commands to create new segments are dealt with first, then the segments of the old batch tape are processed in the order in which the 'update' commands occur. The user can obtain maximum efficiency of the run by ensuring that the 'update' commands are given in the order in which the segments occur on the old batch tape, that commands relating to files are given in the order in which they occur in the segment and that data files follow the program files with which they are associated.

The commands relating to any segment fall into three sections, the edit and input section, the output section and thirdly the execute section. In phase 1, all commands are input, those in sections 1 and 2 are obeyed and a new batch tape is created. The execute commands for phase 2 are added to the directories of the segments (on the new batch tape) to which they apply by STAR. At the end of phase 1, STAR displays a message instructing the operator to load a compiler batch tape on handler 1 in place of the STAR systems tape. This tape may already be held on a spare handler which can simply be selected as handler 1. If the compiler batch is on the same reel as the STAR phase 1 batch then no reloading is necessary.

3.2 Phase 1

Each command is displayed on the output writer before it is carried out, so that the operator can follow the progress of the run. All other messages from STAR are printed on a new line and are preceded by 3 asterisks.

When each segment has been processed, its files are listed on the lineprinter, providing a record for the user. The format of the output is "file name S version number". Similarly, at the end of the run, the contents of the batch directory are listed (only the segment names appear in in this list).

STAR does not change the contents of the old batch tape so it may be processed more than once.

3.3 Phase 2

An execute command causes the specified file to be translated and run (possibly more than once, see 5.4), thus it can be subdivided into at least 2 stages. In the first stage the program is translated, in the second it is run for the first time, in the third it is run for the second time, etc.

Before each stage is carried out, STAR displays sufficient information for the operator to know what is going on. e.g. If the command is

```
FORTRAN,3;testA;1;dataA.
```

then STAR displays the following.

"***FORTRAN 0003 testA 0000" before stage 1.

i.e. Enter FORTRAN at its 3rd entry point to translate testA version 0

"***testA 0001 dataA 0000" before stage 2.

i.e. Enter testA at its first (and only) entry point, use dataA version 0.

If the command is SAP2;4;testS,10,20;1;data1;data1/1.

then STAR displays

```
***SAP2 0004 testS 0000 before testS is assembled, and
***testS 0001 data1 0000 before running testS, and
***testS 0001 data1 0001 before rerunning testS.
```

4. The STAR Commands Tape

The commands relating to a segment are grouped together, and are introduced by a 'new' or an 'update'. They are divided into three parts: an editing/input section, which is carried out first to produce a set of files on the new batch tape, an output section and lastly an execute section during which programs are translated and run.

Any segments, or files within a segment, that are not listed for editing are copied as they stand on to the new batch tape (unless 'nocopy' is in force). If there is no 'update' command for a segment and 'nocopy' is in force then the segment is removed from the batch directory.

Each command begins with a command word (either a basic STAR command or the name of a compiler), and ends with a full stop. The command word is followed by a (possible null) reference set, whose elements are separated by semicolons. Each reference element is either a single name, or else a list (possibly null) with its elements separated by commas. The user must write input/edit commands before the output commands for a particular file.

The last command must be followed by]. All page layout characters are ignored by STAR and may be used to give a neat print-out of the commands tape. The following characters may appear in the commands tape blank, L̲, T̲, S̲, E̲, +   All ignored.

( ) , * / 0 to 9 - . ; A to Z a to z

V̲ is allowed only after an error has been detected during input of the commands tape.

All other characters cause error 1 to be displayed (see 8.1).

The storage space used by a command is approximately one word for each element in the command.

5. Table of Commands

5.1 Introductions (Phases 1,2)

1. batch;batch number, compiler batch name.

This is the first command on the tape. The'batch number'is the number of the reel containing the old batch (zero if no old batch tape). STAR gives a number one greater than this to the new batch tape. The 'compiler batch name'is the name of the relevant phase 2 batch.

2. new;segment name.

This introduces a new segment which is to be added to the batch.

3. update;segment name.

This introduces the commands which refer to an existing segment that is to be modified in some way.

5.2 <u>Edit/Input</u> (Phase 1)

4. nocopy.

This suppresses the automatic copying across of unmentioned files. If this command appears before any of the segment introductions, it applies to all segments of the batch; if it appears among the commands with a particular segment, it applies only to that segment.

5. merge;file list, file name.

The elements in the file list are separated by commas. This creates a new file called file name (the last name), by joining together the character files in 'file list' in the order in which they occur in the list. Files from the old batch tape are merged together on the new batch tape. The new file name may not be followed by a version number. The new file is given version number zero unless the name is the same as one of the files in the list. In that case it is given the same number as that file.

6. inputc;file name.

This inputs a character file from TR1, writes in onto the new batch tape and enters it into the segment directory. If a file of this name already exists in the segment, its version number is adjusted in the directory. If the program (or data) is punched on more than one tape, then each tape except the last must be terminated by & followed by H which cause '***WAIT' to be displayed. The last tape must be terminated by H only. & on its own is ignored. H and & are not stored in the character file.

7. edit;file name.

This writes on to the new batch tape an edited version of the file from the old batch, taking the corrections from TR2. The new file is called "file name" and the previous versions are renumbered accordingly. Editing instructions are given in the style of 503 Library program EDITALL, and must be terminated by an RE instruction. A ST instruction causes '***WAIT' to be displayed. Editing continues when the sign of the word generator is changed.

8. inedit;file name.

This is a combination of edit and inputc. The file that is written on to the new batch tape is the result of editing the information from TR1 with the correction from TR2. The edit instructions must end with an 'RE'. If the program is punched on more than one tape, each one except the last must be terminated by & and H which cause '***WAIT' to be displayed. The last tape must be terminated by H only.

9. list;file name.

This outputs on the lineprinter the contents of the specified file from the old batch tape. Upper case letters have | (vertical bar) printed underneath. Underlined upper case letters have | only printed underneath, i.e. the underline is omitted. Underlined lower case letters are still underlined.

10. copy;file name.

This causes the named file to be copied across to the new batch tape and is used to override the local or global effect of 'nocopy'. The filename is removed from the store copy of the old segment directory.

11. delete;file name.

This suppresses the copying of the specified version of the file from the old batch tape to the new, and removes its name from the old segment directory. It does not prevent the file from being operated on to produce a file for the new batch tape i.e. edited or merged. The version numbers of previous versions are reduced by 1.

12. rename;name list.

This can refer to files or segments. The name list contains 2 names, n1 and n2 say. The file/segment on the new tape that is called n1 has its name changed to n2. Previous versions of the file n1 are not renamed, but the version numbers are reduced by 1.

If the user wishes to rename a segment then he must write the command before the first 'new' or 'update' command on the tape, and if the segment is to be processed then he must quote the name in the update command. The command to rename a file may occur anywhere in the commands relating to the segment but it must precede any commands to output the file.

NOTE 1. The edit instructions for one file must be punched on one tape only.

2. When STAR detects & followed by H it displays '***WAIT'. The operator must load the next tape in the reader and change the sign of the word generator to continue. STAR waits if H is the first non-blank character input in an inputc or inedit command but usually H on its own means end of program and does not cause STAR to wait. If 2 H's are punched at the end of a program, the first one means end of program, the second one causes a wait when STAR obeys the next inputc/inedit command.

3. The edit string in a FL instruction may not contain more than 50 characters (excluding spaces, tabs and blanks).

4. Blanks and erases are ignored by inputc and inedit commands.

5.3 Output (Phase 1)

outc;filename.

This outputs the named file from the new batch tape to TP1. The file is copied across to the new batch tape automatically if it is not already there.

5.4 Execute (Phase 2)

compiler name;trigger pt;file name;trigger pt;data filename.

The program 'file name' on the new batch tape is translated using the compiler specified. The user must ensure that the file (and the data file) is on the new batch tape before the end of phase 1, otherwise STAR may cause the wrong version to be translated or give an error indication if it cannot find a file of that name. The above format applies only to the SAP2 and FORTRAN compilers. For ALGOL the format is as follows:

ALGOL;file name.

If translation is successful, the program is run with data from the new batch tape. For the SAP2 and FORTRAN compilers PCP may be used to read the blocks of data into store, the program must provide its own decode facilities. The method of storing characters on magnetic tape used in STAR is described in 11.1. For the ALGOL compiler STARCH may be used to supply characters, entering it using Elliott instructions, see 11.2.

The trigger points of the compiler and program must be specified. If translation is not successful, STAR causes the compiler batch to be reloaded and obeys the next command.

The file and data may both be qualified by version numbers. In the latter case only 4 versions of the data file are allowed, versions 0,1, 2 and 3.

If it is a SAP program which is to be translated, the estimates required by SAP2 must follow the program name (and the version number if quoted).

```
i.e.   SAP2;trigger pt;file name,x,y;trigger pt; data.  or
       SAP2;trigger pt;file name/number,x,y;trigger pt;data.
```

where x and y are the required estimates.

To translate a program but not run it, the command should be terminated after the file name.

```
i.e.   compiler name;trigger pt;file name.
```

This is meaningful only when the compiler in use is SAP since the FORTRAN compiler deletes itself after translating a program. The user may cause up to 5 SAP programs to be translated consecutively, then he may enter the last program which calls on the earlier ones as common programs, possibly. STAR replaces SPAN basic and SAP by SPAN (complete) before running a SAP program.

To run a program without data, the data name may be omitted but <u>the correct number of ;'s must precede the full stop</u>.

i.e. compiler name;trigger pt;file name;trigger pt;.

Alternatively the program may be run using data loaded in an external source

i.e. any device other than MT2. The data name should be replaced by *.

i.e. compiler name; trigger pt; file name; trigger pt;*.

To run the program with both internal and external data then 'data name*' should be written in the command. The * must follow the data name.

i.e. compiler name;trigger pt;file name;trigger pt;data*.

Before the program is run STAR tests for external data and displays a message to the operator to load data. The user must inform him what to load and in which device.

The same program may be run and rerun 6 times. For each rerun "trigger pt;data name;" should be added to the command.

e.g. compiler;trigger pt;file;trigg pt;data;trigg pt;data.

Before each program is translated the whole of the main and backing store is reset, i.e. the compiler batch is reloaded from handler 1

(For the ALGOL compiler only the main store is reset). This ensures that if one program goes wrong and corrupts the store it does not hinder the translation and running of later programs.

The results of programs run under the control of STAR phase 2 may not be filed in the new batch tape.

The last thing STAR does before transferring control to a compiler/translated program is to position handler 2 in such a way that the next block read is the first block of the mnemonic program/data respectively. It is the responsibility of the compiler/program to ascertain the number of this block as a precaution against the handler being moved.

If a running program reads data from the new batch tape then it must either use PCP to read the data or increment the contents of location 187 by 1 for each block read.

STAR recognises the following as being compilers:

SAP2, ALGOL, FORTRAN.

The entry points to compilers which must be used in a STAR run are:

SAP2 - 4

FORTRAN-3 (without checks), 6 (with checks)

For ALGOL, to compile with checks the compile name should be followed by an asterisk, e.g. ALGOL*;........

The compiler name may be followed by an issue number. This is for information only. STAR cannot check that the right issue is used.

A program returns control to STAR with the STOP instruction.

5.5 EXAMPLE 2

The typewriter output from STAR when it obeys the first commands tape given in EXAMPLE 1 would be as follows:

```
***new reel number 1
batch;0,FORT.
new;seg1.
inputc;testS.
inputc;dataS.
new;seg2.
inputc;testA.
inputc;dataA.
***block no. n CCP m (information needed to repeat phase 2, see 9.5)
***load compiler batch
***WAIT
***FORTRAN 0003 TestA 0000   (translate testA)
```

```
***testA 0000 dataA 0000      (run testA)

***end of batch run
```

The lineprinter output (all in phase 1).

```
seg1          )
testS 0000    )
dataS 0000    )       contents of segment directories
seg2          )
testA 0000    )
dataA 0000    )

seg1          )       contents of batch directory
seg2          )
```

6. Listing of Files

The 'list' command (see 5.2.9) provides a means of producing on the lineprinter a listing of files from the old batch tape.

The command words inputc, edit, inedit and outc can each be followed by an asterisk to indicate that the file that is being written on to the new batch tape is to be listed on the lineprinter at the same time.

7. Error Detection in STAR

The STAR system as so far described assumes that a set of programs will be edited and later run one after the other, each user program returning control to RAP MT, with a STOP instruction when its execution is complete. RAP MT then searches for STAR and re-enters it to obey the next command.

But suppose a user program goes wrong or causes an error interrupt or exceeds its time and has to be stopped by the operator. He must bring control back to RAP MT and take one of the following courses of action:

(i) Resume the run by typing CONT. or CONT;ERRINT.

(ii) Reload the STAR phase 2 batch and go to obey the next command by typing IN, 'batchname'. For ALGOL runs only the run may be resumed by typing STAR.

(iii) If there is a postmortem program available he could enter it and output the current values of the program's data/arrays. Then he can go on to obey the next command as in (ii).

STAR may also detect an error at a stage when one of its subprograms is in control of the computer (during phases 1 and 2). The following paragraphs consider the possible circumstances.

At the start of the run the commands tape is read in; each command is checked for errors in its syntactic structure.

At this stage STAR can refer to the batch directory, and so can also make a check on the segment names that occur in commands. A full semantic check is not possible because the individual segment directories are not conveniently available. If an error is found, the offending command is displayed together with an error message. The computer then waits for the operator to type in a correct version of the command. When this is accepted, the run continues as if no error has occurred. If the operator is unable to correct the error, he types the failure symbol V̲ and a full stop. In this case, the remainder of the commands tape is read and checked, but the batch is not processed. Input stops each time STAR detects an error; to continue the operator must type V̲ . .

No command may occupy more than 13 words in store. In the case of an execute command, the user is allowed to run a program 6 times in the same command. The file list in a merge command may contain at most 11 names, but it must be born in mind that version numbers occupy one whole word, so if any file is qualified by a version number it reduces the total number of files allowed.

In general, if an error is found when STAR is obeying a command then that command is abandoned and STAR goes on to obey the next one. See 'errors in commands' for full details.

8. Error Indications

When an error is detected '*** error n' is displayed and if it occurs during the input of the commands tape then the offending command is output as well.

Phase 1 consists of 2 stages: (a) inputting the commands tape and (b) obeying the commands. In the following error table the phase (1 or 2) and stage (a or b, phase 1 only) in which the error might occur is indicated.

8.1 Table of error numbers

Error No.		Cause	Phase
1	(i)	incorrect format of commands tape	1a
	(ii)	wrong character on commands tape	1a
	(iii)	more than 12 digits in a number on the commands tape	1a
	(iv)	more than 5 SAC programs assembled consecutively	2
	(v)	A 'translate only' execute program given when the compiler is ALGOL or FORTRAN	2
2	(i)	unknown command i.e. first word of a command is not any of the following: 1. A STAR basic command word) 2. A compiler name)	1a

Error No.		Cause	Phase
2	(ii)	None of the compilers in store	2
	(iii)	STAR not in store	2
	(iv)	wrong type of command e.g. give the command copy among the commands for a 'new' segment	1b
	(v)	SPANSWAP NOT IN STORE (SAP run only)	2
3	(i)	Unknown file	1b,2
	(ii)	Version number >7	1b
4		Unknown segment	1a & b
5		Incorrect or insufficient edit instructions for and edit/inedit command	1b
6		1st command not batch	1a
7		Old batch tape has been overwritten, i.e. one (or more) block is the wrong length	1a,& b,2
8		EOT marker passed on input	1b,2
9	(i)	Stored commands list has been corrupted	
	(ii)	Error in format of a command not detected during input	1b
10		EOT marker passed on output	1b
12		PCP magnetic tape function not allowed (BOT)	1a & b,2

Error No.	Cause	Phase
13	PCP magnetic tape function not allowed (EOT)	1a & b,2
14	Old segment directory corrupted	1b

8.2 Other error messages are:

***batch n ***reel m	The batch number, n, given on the commands tape is not the same as the reel number, m, of the old batch tape. The operator must load either the correct reel or the right commands tape and start again by reinputting RAP MT and typing IN; batchname.
***no <device name> End	A device required by STAR is not available. The operator must remedy the situation if he can and start again (Phase 1).
***NODICE End	A PCP error has occurred at the end of phase 1 of the batch run when STAR attempts to cancel its devices. The run stops. (Phase 1).
***NORUN End	Either an error has been found in the commands tape or continuation is not possible. This message is displayed after errors 12, 13 among other places (Phases 1,2).

'***NOTRUN'	A translation error has occurred and the program cannot be run (Phase 2 SAP2 and FORTRAN only). STAR goes on to the next command.
'end of tape on Hn'	This message follows errors 8,10. Phase 1 only. The directory of the current segment is listed and the batch is closed i.e. the contents of the batch directory are listed and it is written on to the new batch tape. The current segment is omitted. (Phases 1,2).
'***<device name and number> on manual'	
***WAIT	The operator should take the device off manual and change the sign of the W.G. to continue (Phases 1,2).
***PARITY END	A parity error has been detected, and still persists after six re-reads of a block during phase 2 (ALGOL compiler only).
I	An identifier has been typed instead of a number when running phase 2 without first running phase 1 (see 9.4, 9.7(b)). The operator must type both the required numbers again.
STARCH ERROR	This is displayed by Starch if it cannot read the program from magnetic tape. It is caused by one of the following.

(i) Parity failure

(ii) Short/long block read

(iii) Block with incorrect number read (not in ALGOL run)

***PCP prog error type n End	This follows a PCP error exit class 2. The run stops (Phases 1,2).
***<file name> absent	A program or data file specified in an executive command is not on the new batch tape when STAR closes the current segment. STAR omits the command and continues.
***PCP R(W,P) err End	A PCP error class 3 has occurred i.e. a parity error while Reading, Writing or Wrong Parity specified. The run stops. (Phases 1,2).
***Hn write permit ***WAIT	Writing is not permitted on handler n. The operator should correct this state and change the sign of the W.G. to continue (Phase 1).

PPPPP.......	This is displayed by the RAP MT magnetic tape input routine. It means that RAP MT cannot read in the first block on the magnetic tape. Interrupt and type IN;NAME. to resume the run if possible.

8.3 Closing the Batch Tape

When the 1st command has been obeyed the current segment is closed, and all unmentioned segments are copied across unless nocopy is in force. If an 'error 7' occurs while the segments are being copied then either

(i) If the old segment directory has been overwritten, the whole segment is omitted and 'segment name removed' is output on the lineprinter. STAR copies the remaining segments.

or (ii) If a block in a file has been overwritten, that file is omitted. STAR copies the remaining files and segments.

8.4 Errors in Commands

In general, if an error is detected in an individual command then that command, is omitted. The exceptions to this rule are listed below.

(i) update.

If an error occurs while closing the previous segment, i.e. a block in one of the files has been overwritten, then the file is omitted and STAR copies the remaining files. If the old segment directory has been overwritten then the segment cannot be processed. STAR ignores all commands until it finds the next 'update' command. 'Segment name removed' is output to the lineprinter and the next segment is updated. If the commands list is corrupt (error 9) STAR attempts to obey the next command.

(ii) merge,inputc,edit,inedit,rename (applies to files).

The command is discontinued. All commands relating to the character file that would have been produced are omitted, i.e. outc, execute.

If the command is edit then the old file is copied across and '***file copied' is displayed.

If the command is inputc, edit or inedit then STAR displays '***WAIT' and waits for the operator to unload TR1 and/or TR2, and to load the next tape in TR1 (and TR2 if appropriate) if any. STAR continues when the sign of the W.G. is changed.

If the command is merge or edit and the error occurs before STAR can determine the version number of the resulting file then it assumes it to be zero.

(iii) Execute.

If the program is not translated successfully, then STAR displays '***NOTRUN' and goes on to the next command.

9. Operating Instructions

9.1 Device Requirements

STAR requires the following devices

(i) tape readers 1,2 (phase 1).

(ii) tape handlers 1,2,3 (phase 1), handlers 1,2 (phase 2).

(iii) lineprinter (phase 1).

(iv) typewriter (phases 1,2).

(v) punch 1 (phase 1).

N.B. Handler 2 remains booked to STAR during the whole of phase 2, so a program must not book H2 although it may read data from the handler.

9.2 Preparing the STAR Systems Tape

All the batches must be made using DUMP2 and all magnetic tapes must be prepared before any batches can be dumped on them (see 2.2.3.4b). In the following operating instructions it is assumed that the magnetic tape (or tapes) has been prepared. STAR issue 2 must be used with DUMP2 and RAPMT.

First, load a tape on handler 1 with writing permitted.

1.	Either input RAP or type	RESET. if RAP is already in.
2.	Input SPAN	IN.
3.	Input SETCBS	IN.
4.	Type	SPAN;5.
5.	If N = no. of units of backing store, type	SETCBS.N.
6.		RESET.
7.	span (basic)	IN.
8.	SAP2	IN.
9.	SPANSWAP Issue 2	IN.
10.	N has the same value as in 5, type	span;N.
11.	Assemble PCP	
12.	Assemble STAR (phase 1)	issue 2.

13.	Attach STAR to PCP	PCP.STAR;2.
	'STAR PHASE 1' is output to the typewriter.	
14.	DUMP2	IN.
15.	Dump the main and backing store	DUMP2.STAR.

This batch may be given any name. If more than one unit of backing store is fitted, one need only specify one unit in steps 5 and 10 and dump the batch by typing DUMP2,6.16.STAR.

9.3 Prepare the Compiler System Tapes

First load a tape on handler 1 with writing permitted.

A different tape may be used for each compiler batch or they may be grouped on one or more tapes.

a. SAP

1-9 Same as 9.2 except than SSspan is used instead of span (basic), and STARCH must be input before SPANSWAP.

Step 10. Input any other binary programs required during the compiling or running of a program. e.g. Read and Print routines for a SAP program.

11.	N has the same value as in 5, type	span;N.
12.	Assemble PCP.	
13.	Assemble STAR (phase 2)	
14.	Attach STAR to PCP	PCP.STAR;2.BATCHNAME.

'***STAR PHASE 2' is output to the typewriter.

15.	DUMP2	IN.
16.	Dump main and backing store.	DUMP2.SAP2.

The batch names typed in steps 14 and 16 must, of course be the same. This batch may be given any suitable name. Notice that SSspan and SAP2 are left in main store. STAR enters SPANSWAP before running a SAP program.

(b) FORTRAN

Follow the same procedure as that for making the SAP batch but use span (basic) and not SSspan. Omit STARCH. After Step 11 assemble the compiler and the DRS. The FORTRAN DRS must be assembled before the FORTRAN compiler.

(c) ALGOL

For ALGOL two batches are required on the same tape called X and Y, say. There may be other batches on the tape as well as the ALGOL ones, e.g. the STAR phase 1 batch.

(i) The actual compiler batch called X, say.

1.	Type	RESET.
2.	Input STAR (Trigger)	IN.
3.	Input STARCH	IN.
4.	Input ALGOL tape 1	IN.
5.	Input ALGOL tape 2 'swait'displayed.	change sign of word generator
6.	Input MOD5 & MOD 10	IN.

N.B. Any other MOD required may be input if it does not interfere with STAR.

7.	Interrupt and type where Y is the name of the second batch containing STAR (phase 2 for ALGOL).	STAR;2.Y.
8.	Input DUMP2. (64 buffer)	IN.
9.	Type	DUMP2;2.X.

(ii) STAR control batch, called Y, say

1.	Type	RESET.
2.	Input STAR phase 2 for ALGOL	IN.
3.	Type	STAR;2.X.
4.	Input DUMP2	IN.
5.	Type	DUMP2;2.Y.

Note that in the entries to STAR (STAR;2) the names of the two batches are exchanged - this is to facilitate interchange between the two batches without operator intervention. The names given to the batches are not significant.

9.4 Other Messages Displayed by STAR

'<SEGMENT NAME> REMOVED'

This message appears on the lineprinter. The segment in question has been removed from the batch directory. e.g. if nocopy is in force and the only command relating to a particular segment is "update;segment name." then that segment is removed.

```
***new reel number N
```

N is the reel number of the new batch tape.

```
*** block no n CCP m
*** load compiler batch
***WAIT
```

This is displayed at the end of phase 1. Phase 2 starts when the sign of the W.G. is changed. The numbers n and m are the block number of the 1st segment directory and the initial value of the current command pointer (CCP). If the operator wishes to repeat/restart phase 2 he has to type these two numbers (see 9.5).

```
***block no n CCP m
***WAIT
```

This is displayed during phase 2, if key 1 is depressed or if entry 5 is made (SAP2 & FORTRAN compilers only, entry 5 is not in STAR for ALGOL). The numbers n and m are the block number of the segment directory and the value of the "current command" pointer - see 9.9 for further details.

```
***load data
***WAIT
```

STAR displays this before running a program with external data.

```
'***compiler/prog.name;trigger pt;file/data name;version no.'
```

This is displayed before a program is translated/run.

```
***nm
```

These numbers were displayed at the end of the original phase 1 run or when the phase 2 run is interrupted by depressing key 1. The former values of n and m must be typed if the user wishes to run the whole of phase 2 and the latter numbers if he only wishes to run phase 2 from the point at which it was interrupted.

***WAIT

(i) This is displayed by STAR when an inedit or inputc command is being obeyed in phase 1 if it reads a 'ST' instruction.

(ii) It is displayed at the end of an edit or inedit command if the edit failed in order that the operator may remove the erroneous edit instructions tape from TR2 before the next command is obeyed.

(iii) It is displayed when the end of an intermediate tape is detected (by STAR reading & and H̲) to allow the operator to load the next tape in the reader.

(iv) It is displayed if H̲ is the <u>first non-blank</u> character read by STAR in an inputc or inedit command. It is suggested that the last tape to be input should be terminated by 2 H̲'s. The first signifies end of program, the second causes STAR to wait before obeying another inputc or inedit command if the operator has not had time to reload the reader(s).

***file copied

An error has been found in an edit command, the old file has been copied on to the new batch tape.

***end of batch run

At the end of phase 2, control returns to RAP MT.

***DUMPIT

Key 1 has been depressed during phase 1. The contents of main and backing store may be dumped on a prepared tape using DUMP2 in order that the STAR run can be stopped now and resumed at a later date. See 9.8.

***SETWG

Set the B-digit of the word generator to compile an ALGOL program with checks. (Phase 2 ALGOL only).

***CLEARWG

Clear the B-digit of the word generator to compile an ALGOL program without checks (Phase 2 ALGOL only).

9.5 The STAR Run

1. Load STAR systems tape on H1, protected
   Load a new reel of tape on H2, writing permitted
   Load the old batch tape, if any, on H3, protected.

2. Input RAP MT (a special version of RAP). I.I's
   Protect the reserved area

3. IN;NAME.

Where 'NAME' is the name of the STAR phase 1 batch.

Now the STAR batch is loaded into main and backing store. While this is going on the operator must load the commands tape in TR1.

4. <u>When the store is set up, control is transferred to STAR automatically and the commands tape is read in.</u> The commands are obeyed one by one (except for execute commands which are stored on a new batch tape ready for phase 2). The operator has only to keep the tape readers loaded (for inputc, inedit and edit commands), collect any output from punch 1 and keep the lineprinter loaded.

5. At the end of phase 1, STAR displays the message

```
'***block number n CCP m'
***load compiler batch
***WAIT.
```

The user must inform the operator in his operating instructions which compiler batch tape he is to load on handler 1. When he has done so, he must change the sign of the word generator and phase 2 will commence automatically. The new batch tape on H2 must be at B.O.T. at the start of phase 2. It is rewound at the end of phase 1 and <u>must not be moved</u>. The compiler batch and STAR phase 1 batch may be on the same reel.

6. During phase 2 programs are translated and run. The operator must load data in devices specified by the user, and collect results from devices named by the user.

STAR requires the typewriter and handlers 1 and 2 only; the lineprinter punch 1 and handler 3 are available to the user.

Phase 2 cannot be run without first running phase 1 once to prepare a STAR batch tape containing the desired programs and executive commands, also to ascertain and display the values of n and m (see 9.4).

However, it is possible to repeat or restart phase 2 by pressing the message button, re-entering STAR phase 2 at its 4th entry point and typing in the block number, n, and initial value of the current command pointer, m, which were displayed at the end of phase 1. This has the effect of setting pointers back to the values they held at the start of phase 2, transferring to the magnetic tape input routine in RAP MT to reload the compiler batch and restart phase 2.

The operator must type

```
PCP;3.STAR;4.n.m.
```

When using the ALGOL compiler, it may be necessary to reload the STAR phase 1 batch before making this entry. Phase 2 can be restarted, thus any number of times.

9.6 Continuation after errors

	Error	Action
(a)	Phase 1	
1.	Error in the commands tape detected while it is being input. The erroneous command is displayed.	Type either a correct version of the command, including a full stop, or the failure symbol V̲ and full stop.
2.	'error message' 'End'	Continuation is not possible. See 8.2 for a list of error messages.
3.	All other error messages not followed by 'End'.	STAR attempts to continue the run.
(b)	Phase 2	
4.	'***error 2'.(SAP2 & FORTRAN runs only) The wrong batch tape has been loaded on H1 i.e. no compilers in it/STAR missing/ SPANSWAP missing (SAP run only)	Load the correct reel on H1 and type IN; BATCH NAME.. The compiler batch is reloaded and the same command obeyed again.

	Error	Action
5.	The batch on H1 contains the wrong compiler. e.g. Trying to assemble SAC programs using FORTRAN. This should be obvious to the operator from the typewriter output, but STAR does not detect it.	As in 4 above.
6.	When ALGOL compilation errors are detected - the compiler goes to SWAIT	It may be necessary to press the message button and type STAR. to continue the batch run. If STAR is corrupt then the STAR phase 2 batch should be loaded by typing IN;NAME. where NAME is the name of the batch containing STAR phase 2.
7.	When excessive FORTRAN compilation errors occur.	Interrupt and reload the batch to continue.

9.7 Entries to STAR

(a) Phase 1

1. Dummy entry. If control returns to RAP MT during a STAR run then RAP MT transfers to STAR's first entry point which causes control to return to RAP MT immediately.

2. <u>Attach STAR to PCP</u>. This entry must be made when preparing the STAR batch tape. See 9.2.

3. <u>Main entry to phase 1</u>. DUMP2 transfers control here after reading the STAR batch into main and backing store.

4. The facility of stopping at the end of the current command is now under keyboard control (see 9.8). Entry 4 now becomes a dummy entry as in entry 1.

5. <u>Go on to the next command</u>. The current command is abandoned (unless it is 'new' or 'update' which must be completed) and the next one obeyed. To use this entry the operator must interrupt the STAR run by pressing the message button, and typing PCP;3.STAR;5.

6. <u>Repeat this command</u>. The current command is restarted (unless it is 'new' or 'update' which may not be repeated). The mode of entry is as in entry 5.

(b) <u>Phase 2</u>

1. <u>This entry is used by RAP MT</u>. Each time control returns to RAP MT from a compiler or running program by means of a 'STOP' instruction, RAP MT searches for STAR and re-enters it at this entry point.

2. Attach STAR to PCP and save the batch name in STAR's workspace. This entry must be made when preparing a compiler batch tape. See 9.3. In the case of FORTRAN or SAP2 compiler batches, the name of the batch must be typed so that the batch can be reloaded by RAP MT.

   When running ALGOL programs, there are two phase 2 batches and each must know the name of the other one so that each can load the other using RAP MT and without operator intervention. Entry 2 to STAR (trigger) and STAR (phase 2 for ALGOL) must be used to set the batch names. See 9.3(c).

3. Main entry to phase 2. DUMP2 transfers control here after reading the compiler batch into main and backing store.

4. To repeat or restart phase 2, enter STAR phase 2 at its fourth entry point by pressing the message button (if necessary) and typing PCP;3.STAR;4. The block number, n, and initial value of current command pointer, m, which were displayed at the end of phase 1 (see 9.4, 9.5) must then be typed. This entry does not exist in STAR (Trigger).

5. This entry prints out the values of the block number, n, and current commant pointer, m, for the command being obeyed. It should be used if the phase 2 run has to be suspended for any reason by pressing the message button and typing

```
PCP;3.STAR;5.
```

When the STAR run can be resumed, the operator <u>must reinput RAP MT</u>, load the STAR phase 2 batch and after the message '***nm', type in the values displayed by STAR entry 5. The run is resumed, starting with the interrupted command. This entry exists only in the version of STAR phase 2 for SAP2 and FORTRAN.

An alternative method of interrupting phase 2 (all versions) under keyboard control is described in 9.9(b).

(c) <u>To run phase 2 without running phase 1</u>

Phase 1 must be run once to prepare a new batch tape but phase 2 can be run any number of times using this new batch tape. The location in RAP MT which is set by phase 1 will be clear on entry to phase 2 if phase 1 has not been run. Phase 2 displays the message ***nm and waits for the operation to type in the 2 integers n and m (each terminated by a full stop) which were displayed at the end of the original phase 1 run (see 9.4, 9.5.6) or at some time during phase 2 (see 9.9). They provide the information necessary to set up the RAP MT location and start the phase 2 run.

'n' and 'm' may vary from one new batch tape to another, therefore RAP MT must be reinput each time phase 2 is run using a different batch tape on H2.

9.8 Interrupting Phase 1

It is possible to interrupt phase 1 while it is obeying the commands in order to dump the contents of main and backing store. This is useful if STAR is over-running its allotted time. The operator can complete the run on another occasion merely by loading the same tapes on handler 2 and 3 and loading this batch.

STAR will stop after obeying a command if Key 1 is depressed. This key setting has no effect while the commands tape is being input. STAR allocates a SPAN block of 1000 words which is large enough to hold DUMP2 and sets the FF and LF pointers pointing to the first and last words of this block respectively. The tapes on H2 and H3 are rewound. Then '***DUMPIT' is displayed.

The operator should now read in DUMP2 and dump a batch on any magnetic tape available.

When this batch is reloaded the STAR run is resumed automatically. However if the operator wishes to continue the run immediately after dumping the batch he must type

PCP;3.STAR;3.

9.9 Interrupting Phase 2

(a) Operator Intervention

SAP2 and FORTRAN only

Phase 2 can be interrupted by pressing the message button and typing PCP;3. STAR;5. This causes the block number X and current CCP value Y for the current command, i.e. the one being obeyed when the run was interrupted, to be displayed. To continue the run on another occasion the operator should load the correct batch on H2 at B.O.T., input RAP MT and load the compiler batch. In response to the message '***nm' he should type X.Y.. The run is resumed and the first command to be obeyed is the one that was interrupted.

Alternatively he may wish to interrupt one program (e.g. because it is overrunning or has entered a loop) and go on to obey the next command immediately. In this case he should press the message button and type IN; "compiler batchname".
If the FORTRAN compiler is interrupted, the operator must reload the batch to continue.

ALGOL

The operator may wish to interrupt a program (perhaps because it is overrunning) or the compiler (because there are syntactic errors) and go on to obey the next command immediately. In this case he should press the message button and type STAR. . Should STAR be corrupt or overwritten, the operator should reload the STAR phase 2 batch by typing IN; "batchname".

(b) Keyboard control

Phase 2 can be interrupted at any time by depressing key 1. The interruption will take effect after the batch containing STAR phase 2 has been reloaded, and the values of block number, X, and CCP, Y, for the next command are displayed. STAR then outputs '***WAIT' and continues the run when the setting of Key 39 is altered.

To restart the run at this point on another occasion, the operator must load the same tapes on handlers 1 and 2, reinput RAP MT and load the STAR phase 2 batch when '***nm' is displayed he should type X.Y. and the run will continue from the point at which it was interrupted.

SAP2, FORTRAN

To obtain the values X,Y for the current command, entry 5 to STAR phase 2 must be used.

ALGOL

The values of X and Y for the current command are obtained by subtracting 2 from Y, i.e. the block number is the same and the value of CCP is 'Y-2'.

If an ALGOL program is taking longer to run than anticipated, and the operator has to interrupt the run, he should take the following action:

(i) Press the message button

(ii) Depress Key 1

(iii) Reload the STAR phase 2 batch. 2 numbers will be displayed, W and Z say. These values must be preserved.

(iv) To resume the run later he must load the same tapes on H1, H2, reinput RAP MT, load the STAR phase 2 batch and type W."Z-2". after the message '***nm' has been displayed.
The run is resumed and the interrupted command is restarted.
If he types Z instead of Z-2 the next command is obeyed, and not the interrupted one.

10. RAP MT

The use of STRAP has been discontinued. RAP MT replaces it. It contains a simple magnetic tape input routine entered by typing IN;"batchname".

RAP MT will automatically re-enter STAR at its first entry point after a 'STOP' instruction is obeyed and after the message 'NO PROG' is displayed.

Certain facilities present in RAP are omitted in RAP MT.

The following messages can be typed on the typewriter:

CONTROL MESSAGES	EFFECT
IN.	Effect same as in RAP.
CONT.	Effect same as in RAP.
CONT;ERRINT.	Effect same as in RAP.
RESET.	Effect same as in RAP.
LIST.	As in RAP but outputs X after the freestore size.
IN;batchname.	Transfers control to the magnetic tape input routine in RAP MT. This will input the batch loaded on H1.

The messages not available in STRAP are:

IN;N. (where N is an integer)

CANCEL.

CANCEL;name.

FREEST.

N. and N;S. (where N is an integer)

Displayed messages

All the displayed messages in RAP are also in RAP MT (see 2.2.1).

The message END is only displayed when a STOP instruction is obeyed and STAR is not present.

When RAP MT is unable to read the first block on a magnetic tape 'P' is displayed and attempts are made to reread the block. A continuous output of P's when loading a batch tape indicates either a poor handler or poor magnetic tape and the operator should take appropriate action.

11. STORAGE OF DATA ON MAGNETIC TAPE

11.1 Format of storage

The blocks of data are written as SPAN blocks of 122 words. The first word after the SPAN header is for PCP's use and holds the block number in the top 19 bits. The remaining 120 words are free for data. The SPAN headerword is not included in the block written up to magnetic tape. The mode of writing is odd parity, binary mode.

Five characters of 7 bits are packed in each word. The top 4 bits are left blank, the first character occupies bits 35 to 39 inclusive, the second, bits 28 to 22 inclusive, etc. If a haltcode is reached before the block is filled, the rest of the block is filled with zero words. Halt codes and ampersands are not written onto the magnetic tape.

11.2 Access to data

SAP2 programs may use PCP to read down the data blocks at run time(the magnetic tape will have been positioned by STAR so that the first block of data is the next block to be read). It must provide its own unpacking routines.

ALGOL normally uses the common program STARCH to read its program from magnetic tape. In the same way the compiled program may use STARCH as a common program using Elliott instructions to enter it.

STARCH reads one magnetic tape block at a time into the buffer, unpacks each word, and exits to the calling program with the next character in the accumulator.

The form of entry to STARCH required by an ALGOL program (written in Elliott Instructions) follows:-

```
            30  <9STARCH>
            73   7932:44 8032
            42   nostarch:20 save
            04   <+5>     :55 20
            04   <73 0:400>: 10 save
            20   HOLD / 30 1
            51   20       : 22 save
            20   HOLD+1 /  70 4
            03   <04 8191:>:51 20
            04   save     : 20 save
     save)  +0                                'save' is entry1 to set
                                              up STARCH
            32   save     : 20 locn
                                              the normal entry is set
       :                                      in 'locn'.
       :
       :
     HOLD)  +0
            +0
       :
       :
       :
```

12. NOTES ON PERFORMANCE FIGURES

The first speed given for each command is the rate for the operation in the description. The second speed is that of the operation plus listing the file (edited in cases 2 and 3) on a 300 line/min lineprinter.

command	effect	speed in ch/sec	speed in ch/sec + listing
inputc	input from paper tape	970	61
inedit	input with editing from paper tape	570	61
edit	input from magnetic tape, edit from paper tape	650	56
outc	output to paper tape	100	50

Phase 2

The time taken to translate a program from magnetic tape using SAP2 is approximately 866 ch/sec (plus 8 secs for messages to the typewriter from SAP2 and STAR). The time to translate a program from paper tape using SAP2 is 520 ch/sec (with no blank characters) (plus 6 secs for messages to the typewriter from SAP2 and say 3 secs to type the entry to SAP2).

i.e. the rate for STAR is 866 ch/sec + 8 secs

for SAP2 only is 520 ch/sec + 9 secs.

The following command was timed, with two operators who had never operated a STAR run before, and were following operating instructions.

```
batch;o,BATCH3.
new;TEST.
inedit;IMP.
outc*;IMP.
SAP2;4;IMP,90,60.
]
```

The break-down of time was as follows:-

1.	LOAD 3 magnetic tapes	85 secs
2.	LOAD paper tape in reader, set word generator etc.	40 secs
3.	TIME TAKEN FOR STAR BATCH TO BE LOADED (3rd BATCH)	55 secs
4.	Commands are read in-to message ***WAIT	6 mins 10 secs
5.	TIME TAKEN FOR PHASE 2 BATCH TO BE LOADED (3rd BATCH)	55 secs
6.	PHASE 2 total time	23 secs
	TOTAL TIME FOR BOTH PHASES	10 mins 28 secs.

This time could have been reduced by 1-1½ minutes by having the batches for Phase 1 and Phase 2 as the first batch on different reels (steps 3 and 5).

```
batch;2,BATCH 1.
new;DEMO.
inputc;ALG1.                 2
inputc;ALG2.                 5
inputc;ALG3.                 5
update;ACC.
edit;IMP.                   13
copy;EDIT.                   7
copy;PT.                    12
SAP2;4;IMP,90,56.
SAP2;4;EDIT,90,47.
SAP2;4;TEST,8,30;7;PT*.
update;DGF.
edit;FORT3.                  1
list;FORT2.                  1
update;SEG1.
edit;STAR.                  69
update;SEG2.
SAP2;4;q,10,10;1;.
rename;r,p.                  1
list;r.                      1
]
```

The above commands list was timed from the beginning of the command being input to the message ***WAIT at the end of phase 1. The time taken was 180 secs. The number of blocks in each file are listed beside the command.

13. <u>RESTRICTIONS</u>

The minimum command which will copy the old batch tape onto the new batch tape, and update the batch number by one is:-

```
batch;number,compiler name.
update;SEGMENT.
]
```

SEGMENT is any segment in the batch; this will be put first onto the new batch, the remainder will be copied in the order in which they occur on the old batch tape.

Every command tape must have at least 1 update/new.

Appendix 1

CODE STCOPY S

FUNCTION

To be used in conjunction with RAPMT and DUMP MK.2 (Issue 2) to copy STAR batch tapes.

CONFIGURATION

As for STAR.

METHOD OF USE

STCOPY can be used to copy batch tapes for STAR/FORTRAN, STAR/SAP2 and STAR/ALGOL1, provided that the tape has been prepared using DUMP MK2. Issue 2.

OPERATING INSTRUCTIONS

The instructions given below are for STAR/ALGOL1. Reference should first be made to 503 MANUAL, 2.5.5.9.

The 3 batches on the STAR/ALGOL1 batch tape are called STAR1, STA2.X, STA2.Y in this description. The batch tape to be copied is called BT1 and the tape it is to be copied onto BT2.

Step No.	TAPE	OPERATOR ACTION	RESULT
1	BT1, no write permit BT3, write permit	load to any two handlers	
2		RAPMT in store	
3	STCOPY (binary)	IN.	
4		STCOPY.	END

Step No.	TAPE	OPERATION ACTION	RESULT
5	BT2, H1 remote	RESET	
6	DUMP2	IN.	
7		DUMP2;5.	Header written on BT2 END
8	BT2, H1, remote	IN;STAR1.	END
9	DUMP2	IN.	
10	BT2, H1, remote	DUMP2; 6.16.STAR1.	batch written to BT2 END.
11	BT1, H1, remote	IN;STA2.X.	END
12	DUMP2	IN.	
13	BT2, H1, remote	DUMP2;2.STA2.X.	batch written to BT2 END
14	BT1, H1, remote	IN;STA2.Y.	END
15	DUMP2	IN.	
16	BT2,H1, remote	DUMP2;2.STA2.Y.	batch written to BT2 END
17		re-input RAPMT	

OPERATING NOTES

(1) The copied batches must be given exactly the same names as they had on BT1.

(2) It will probably be convenient to have three copies of DUMP2 on a single paper tape.

(3) The version of DUMP2 used must have a 64 (or less) word buffer, otherwise it will fail to read in at step 10.

(4) If BT2 had already been prepared with DUMP2, steps 6 and 7 may be omitted.

(5) STCOPY corrupts RAPMT, so that step 17 should never be omitted.

TAPES

No program tape is issued but the text of the program is as follows:-

```
program   STCOPY;
block     one;
begin     one;
          30 <30A : 20 8108> (load order pair to overwrite
                               9 STAR)
          73 7932 : 44 8147 (enter RAP to obey order pair
                              in acc)
          STOP
      A)  80 65 00 00 00 00 00 (illegal RAP name 90)
end;
trigger   one;
```