FOREWORD

OPIT was a program for closed-shell self-consistent field, molecular orbital calculations using a basis set of floating spherical gaussian functions. The attached paper, from the early 1970s, illustrates very clearly the efforts needed to conduct calculations of this sort, even for small molecules, on the mainframe hardware available at the time. Elapsed times of minutes, or even hours, were not uncommon and the paper describes a data overlaying scheme, using hash coding and linked lists of main memory and disc blocks, to ensure that resources on the ICL 1906A were used to the very best effect (there was no virtual memory and the technology of main memory was magnetic core storage).

Younger readers should note the following changes in usage over the years:

core store = main memory;
mill time = CPU time;
backing store = disc storage;
consolidation = link editing;
degenerate hash = overflow hash.

The last-named of these was terminology for the case where two or more distinct file names happened to hash to the same integer value. We hi-jacked this terminology from the group theory of molecules, where certain of the irreducible representations of the molecular point group were doubly or triply degenerate, leading to corresponding molecular energy levels having identical values. Only after the paper was published did we discover that the accepted terminology was ‘overflow hash’.

COLOPHON

The attached paper is a re-built version of the original published paper as properly re-typeset ‘PDF Normal’ rather than just as a bitmap scan.

This particular paper appeared in the journal “Computer Physics Communications” (Elsevier) in 1973. The canonical published version is available online (to subscribers, or for one-off purchase) as a scanned bitmap PDF via: `http://www.sciencedirect.com/`

The text was acquired by scanning the paper from an offprint and then using Readiris OCR on the resulting TIFF files. The paper was then re-typeset using UNIX *troff* suite to set up the correct body typeface (Times) and to get the line and page breaks reasonably accurate. The tables and line diagrams were re-set using the *tbl* and *PIC* pre-processors for *troff*. The lettering in the line diagrams uses the Tekton typeface.

The time taken to build this paper (over several lunchtimes …) to a reconstituted ‘final draft’ form was about 6 hours.

COMPUTER PHYSICS COMMUNICATIONS. 5 (1973) 123–135. NORTH-HOLLAND PUBLISHING COMPANY

THE OPIT SYSTEM

I. A FILE HANDLING SCHEME FOR DATA IN LARGE APPLICATIONS PROGRAMS

John C. PACKER
University of London, Computing Centre, London W.C.1., UK

and

David F. BRAILSFORD[*]
Department of Mathematics, University of Nottingham, Nottingham NG7 2RD, UK

Received 5 April 1972
Revised manuscript received 30 September 1972

The OPIT program is briefly described. OPIT is a basis-set-optimising, self-consistent field, molecular orbital program for calculating properties of closed-shell ground states of atoms and molecules. A file handling technique is then put forward which enables core storage to be used efficiently in large FORTRAN scientific applications programs. Hashing and list processing techniques, of the type frequently used in writing system software and computer operating systems, are here applied to the creation of data files (integral label and value lists etc.). Files consist of a chained series of blocks which may exist in core or on backing store or both. Efficient use of core store is achieved and the processes of file deletion, file re-writing and garbage collection of unused blocks can be easily arranged. The scheme is exemplified with reference to the OPIT program.

A subsequent paper will describe a job scheduling scheme for large programs of this sort.

1. Introduction

For many years now, fast, modern computers have opened up new avenues of exploration in theoretical physics and chemistry. Understandably, however, in the rush to complete calculations which were previously impossible, efficient programming techniques and optimum use of computer resources (e.g., core storage, magnetic tapes) has often taken second place. This paper represents an attempt to redress the balance.

The OPIT program performs self-consistent field molecular orbital (SCF MO) calculations for atoms and molecules using a basis set of spherical gaussian basis functions. The original version of the program was originated and implemented, in ALGOL, by Packer et al. on the University of Nottingham KDF 9 computer [1–3]. The present version of the program has been implemented on the ICL 1900 series computers under the GEORGE 3 operating system. The program code is mainly in FORTRAN but several routines involving bit manipulation have been written using the PLAN 4 assembler.

1.1. *Operating considerations for large programs*

Although there are many well-known programs for performing SCF MO calculations (e.g., IBMOL (4) and POLYATOM [5,6]) a feature common to these programs, and to many others in theoretical physics and chemistry, is the excessive amount of core store required, which makes multiprogramming with other users of the computer well-nigh impossible. The large amount of core required is chiefly the fault of the

[*] To whom all communications should be addressed.

FORTRAN language. Although the object code produced by a FORTRAN compiler is fast in execution, core storage is most inefficiently used since array bounds are static and are fixed at compile time. Attempts to implement dynamic array bounds in FORTRAN can be made to work but are often unwieldy [7}. The result of this is that vast amounts of core are claimed in array declarations so that the largest calculations can be performed. Thus, smaller calculations can still be processed by the same program but at the expense of claiming a lot of core storage that never gets used.

We feel that if a program is to be used many times as a standard package it is worth spending some time and trouble to ensure that core is efficiently used and that backing store is used only when absolutely necessary. This belief led us to implement the file handling scheme described in section 6 onwards. The scheme aims to map files of various sizes onto a fixed amount of core in an efficient way. The characteristic of these files is that their size is not known until run time.

It is not the purpose of this paper to describe in detail the quantum chemical aspects of OPIT since this has already been done [1}. However, we shall give a brief description of the problems encountered in calculations of this type in sections 2–5, so that the need for the file handling scheme can be better understood.

2. Nature of the calculation

The total energy, and other atomic and molecular properties for closed shell ground states, are to be obtained using the SCF MO scheme to solve the Schrödinger equation. The expression of a ground state wavefunction as a single Slater determinant of double occupied orbitals, and the expansion of each orbital in terms of a finite basis set [8,9], are so well known as not to require elaboration here. For computational purposes the Hartree-Fock self-consistent equations are cast in a matrix form and the Hartree-Fock matrix, **F**, is built up at every stage in the iteration process, from a knowledge of the first order density matrix and the integrals of the SCF hamiltonian over the chosen basis set [4–6,9].

Two of the most common choices for a basis set are Slater type atomic orbitals and gaussian functions. Other things being equal, the larger a basis set, and the closer it approaches completeness for the expansion of the orbitals, the more accurate a calculation will be, within the limits imposed by the Hartree-Fock approximation. Our approach,however, is to use a very modest basis set $\{g\}$ of spherical gaussians [2], where

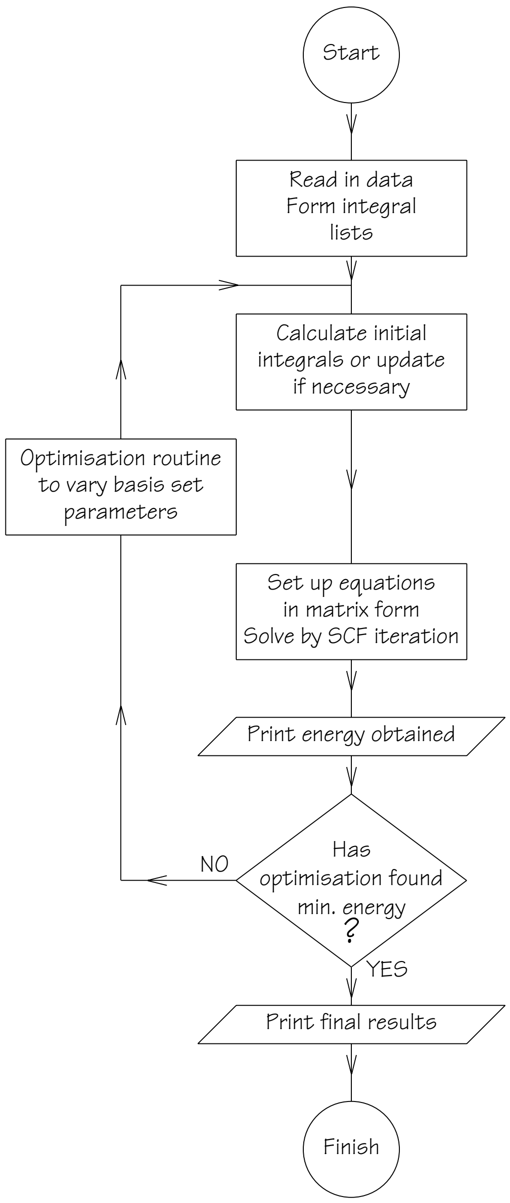
Start
Read in data
Form integral lists
Calculate initial integrals or update if necessary
Optimisation routine to vary basis set parameters
Set up equations in matrix form
Solve by SCF iteration
Print energy obtained
NO
Has optimisation found min. energy ?
YES
Print final results
Finish

Fig. 1. Overall structure of the OPIT program.

$g = N \exp[-\alpha(r - \boldsymbol{R})^2]$

and N = normalization factor, α = gaussian exponent, r = position co-ordinate relative to the gaussian centre, $\boldsymbol{R}$ = position of gaussian centre relative to some fixed origin of co-ordinates.

If the basis set is small, not only can one solve for the optimum set of MO coefficients for a fixed basis using the Hall-Roothaan equations [8,9], but also it is possible to include an optimisation routine, encompassing the whole program, which will systematically find the best position co-ordinates ($\boldsymbol{R}$) and exponents (α) within the given basis set.

A well-known feature of all quantum chemical calculations is the large number of one-electron and two-electron integrals generated by integrating the basis set over the SCF operator. When an optimisation routine is used, in the way described above, it is necessary to have a technique to detect basis functions altered by the optimisation routine, and to recalculate any integrals that will, in consequence, have been altered. Details of these routines are given in [1]. A simplified flow chart for the whole OPIT program is given in fig. 1.

3. The components of the program

3.1. Integral listing

The technique for listing the one- and two-electron integrals is an adaptation of that described in POLYATOM [5]. Briefly, it involves reading in a transformation matrix to describe the behaviour of an indexed basis set under the transformations of the molecular point group to which the set belongs. Using this table, lists of one-electron and two-electron integrals are built up; each entry in the list corresponds to combinations of the basis set indices. The one-electron integral lists have $\frac{1}{2}N(N+1)$ entries for N basis functions. The two-electron integral list has no less than $\frac{1}{8}N(N+1)(N^2+N+2)$ entries. All lists take the form of a series of sublists. A ‘sublist head’ denotes a typical integral of a certain type and it is immediately followed by a ‘sublist body’ which represents integrals that have the same value as the sublist head due to the symmetry of the problem. The number’of sublist heads can be predicted knowing the point group spanned by the basis functions and the nuclei [10].

3.2. Integral evaluation

Integral evaluation for the spherical gaussian basis functions over the various operators contained in the SCF hamiltonian is relatively straightforward and the formulae quoted by Shavitt [11] are used. The electron-nucleus attraction integrals require, for a spherical gaussian basis, a knowledge of F_0 which is closely related to the error function. We have used the highly accurate evaluation described by Shipman and Christoffersen [12] and have found it to be most satisfactory.

At the end of the integral evaluation, files of integral values have been produced. There is a one-to-one correspondence between each integral value and an entry in an integral list.

Values in the files of integrals will have to be altered in response to variation of the basis set parameter by the optimisation routine (see fig. 1). The updating process is designed to preserve at all times, the overall spatial symmetry of the basis set. For this reason the files of integral labels, generated from symmetry transformations, do not need to be altered.

3.3. The SCF process

In this phase each integral labels list, and its corresponding list of integral values, is used to set up the matrices required for the SCF process. The one-electron integrals are dealt with first and four distinct one-electron matrices are set up.

The $\mathbf{F}$ matrix is then set up using these four matrices together with the two-electron integrals. This latter file is suitably processed [5] using the boolean information packed into the label words. The SCF equations are then solved iteratively in the usual way.

At every stage of the SCF iteration it is necessary to transform the $\mathbf{F}$ matrix from the normalised but nonorthogonal gaussian basis to a suitable orthonormal basis. This transformation is effected by the Tcholesky decomposition [13]. The eigenvalues and eigenvectors of $\mathbf{F}$ in the new basis are found using the QL algorithm [14].

After each iteration the total energy is calculated. Self consistency is deemed to have been reached when successive energy values differ by less than 0.00001 hartree. The whole SCF process is repeated many times during the course of optimising the basis function parameters (see fig. 1).

4. Efficiency considerations in integral listing and labelling

The schemes described above are well tried and tested but keeping both integral value and label lists is very wasteful of storage. At the moment we store $\frac{1}{8} N(N+1)(N^2+N+2)$ two-electron integral labels and a corresponding number of integral values. An obvious modification would be to store only distinct integral values (i.e., those which correspond to sublist head entries in the integral labels list). In the present scheme the only point in performing the listing operation is to avoid re-evaluation for those integrals which are the same as a sublist head.

An algorithm, which greatly reduces the storage space required, for integral labels and values, has been proposed by Dacre [15]. However, in this method the building up of the **F** matrix could be a slow process since symmetry information is not used until a late stage. Conversely, an algorithm given by Duke [16] is very efficient for **F** matrix formation, but retains the uneconomical storage features of the present scheme. However, a method which combines these two algorithms would probably be very effective.

5. Optimisation

The optimisation routine used is that due to Powell [17]. This is a direct method−that is, it does not involve a knowledge of the first or second derivative of the function being minimised (molecular energy) with respect to the variables involved (gaussian positions and exponents). None the less, if a package could be developed to calculate these derivatives in the present case, there is little doubt that the minimum energy configuration could be found with considerably fewer function evaluations.

The use of an optimisation routine prompts the use of total molecular energy as a convergence criterion in the SCF phase (see section 3.3) rather than checking convergence of elements of the density matrix. If the SCF process ever fails to converge, an energy value will be produced which is greater (i.e., less negative) than the true value. Such a value, and the basis set configuration which generated it, will be ignored, as the optimisation routine is constantly searching for an energy minimum. Thus, provided the SCF process is stable in the region of the optimum configuration, the optimisation routine should eventually find the correct minimum energy.

6. The file handling scheme

We shall now describe a scheme for file storage and handling which could be adapted for use in any situation where a program ought to make fuller use of the available core and backing store facilities. Although the concept is, in principle, independent of the type of program and the machine on which it is run, we shall exemplify certain points with reference to an implementation of the OPIT program (see sections 2–5) in FORTRAN and PLAN on an ICL 1900 Series computer. The coding of the file handling scheme has been done in FORTRAN wherever possible but for some routines, involving word packing and bit pattern comparisons, we have had to resort to using the ICL PLAN 4 assembly language. For the purposes of this paper an ICL 24 bit storage location is referred to as a half-word. Two of these locations (48 bits) can be used to store real numbers and will be referred to as words.

It will be apparent from sections 2–5 that during the course of running the OPIT program several named files, of integral labels, integral values, matrices etc. are set up. These files must be stored and updated appropriately when necessary. Moreover, there is a large disparity between the size of the smallest file (N^2 entries for a basis set of N members) and the largest ($\approx N^4$ entries). For large basis set problems, it will be necessary to use disc backing store, since the number of two-electron integral labels and values becomes very large. In Packer’s [1] original KDF9 implementation of OPIT a totally disc based file storage scheme was adopted. This is convenient for program structuring, and fairly efficient for large basis set problems, since disc transfers could be overlapped with central processor activity. For small problems, however, the scheme was very inefficient and it would have been desirable to store all files in core.

The file handling scheme described here is sufficiently general for it to be capable of extension, in the event of more files being created, read from or written to. It also has the desirable attribute of running small basis set problems in core wherever possible. Overflow from core to disc as the basis set increases is automatic and almost invisible to the user, thus simulating a ‘one level store’.

The concept of using backing storage, in conjunction with core, to provide a large ‘virtual memory’ is now very familiar [18,19]. The principal methods of using virtual memory are by segmentation or by paging. The advantages and disadvantages of both schemes are discussed in a comprehensive review by Denning [20]. In the former method the program and data space is divided up into variable length segments, often in a manner defined by the programmer. In the latter method the program and data space is mapped onto ‘pages’ of a fixed size. At any moment only a fixed number of pages will be present in core, the remainder being held on a backing store device. The decision as to which pages are currently kept in core is usually done automatically by the virtual memory device of the machine in use−though the decision is normally based on the observed frequency of use of any given page.

The present scheme was developed on a machine that did not have paging hardware and to handle large programs the only option available was to overlay (i.e., segment) the program. The standard ICL overlay scheme [21] works best for overlaying program code, i.e., when the program code is large and its associated in-core data space is small. However, in our case, we have the problem of modest size program code which easily fits into core together with a very large amount of data space. The ICL scheme permits data space to be overlaid only if the appropriate variables are declared in DATA statements. Moreover, although the structure of a program is often fairly transparent, in the way that subroutines are linked together and called by a master program, it must also be appreciated that data often has a structure of its own, which is not immediately apparent from array declarations etc. within the program. For this reason we chose to implement our own ‘overlaying’ of data. By storing the bulk of the program’s data in named files of varying length, and by setting up a file dictionary, it is possible to segment the data and arrange for overlaying of data. as required, into a fixed length workspace area. The scheme will now be described in detail.

7. The file dictionary and core storage array, ISTORE

All file handling and in-core file storage within the OPIT program is carried out in an array ISTORE. The original intention was that this should be of type INTEGER. Unfortunately, it has had to be declared REAL in 1900 FORTRAN to circumvent certain difficulties due to the way integers are represented in 1900 Series machines. ISTORE contains mainly packed words which are manipulated by routines written in machine code (see appendix 1). The nature of ISTORE (INTEGER or REAL) will be immaterial on most machines.

The layout of array ISTORE is shown in fig. 2. By using the PLAN 4 `#ELASTIC` directive [22] one can force the ISTORE array to the uppermost addresses of the program and data area (or field length). This facility is useful when performing postmortem dumps, or if future program enhancements wish to alter the size of ISTORE during execution. The limit pointer for ISTORE is IFL and

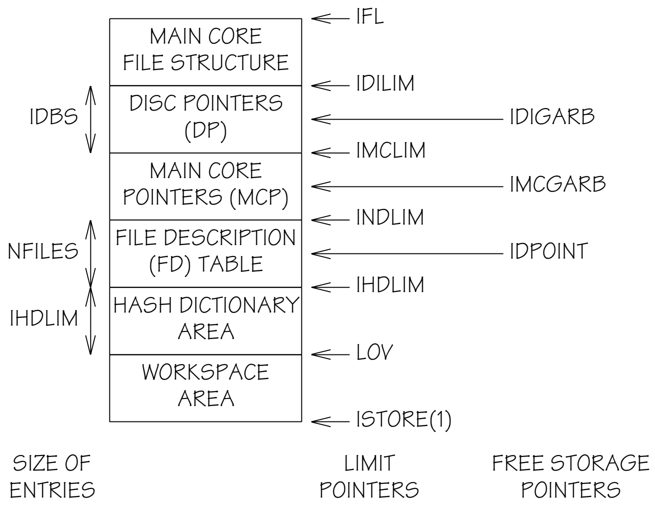

Fig. 2. The layout of array ISTORE.

this also marks the limit of the whole field length. It is not strictly necessary to have array ISTORE at the top of the field length, but it does facilitate changing the size of the array for different runs of the program or even during a run. Hence, programs using this scheme could adjust their core size to suit the current conditions on the machine.

The various areas of ISTORE will now be examined in turn.

7.1. The workspace area

Files of integral labels and values are read down into this area while they are being processed. Matrix setting up and manipulation during the SCF phase (see section 3.3), is also carried out here. Waste of core due to FORTRAN's static array bounds is avoided by carefully placing all matrices and files within the workspace area, whose limit pointer (LOV) is set at run time from a knowledge of the characteristics of the problem to be run. For example, let the maximum workspace areas required by the integral listing, integral evaluation and SCF processes be LINTS, LIVS and LSCF respectively then

$$\mathrm{LOV} = \mathrm{AMAX0}\,(\mathrm{LINTS},\ \mathrm{LIVS},\ \mathrm{LSCF}\,) + 1.$$

Having computed the limit pointer in this way the detailed placing of files within the workspace area is achieved by calculating the limit pointers for each of the files and using these pointers to index the array ISTORE in an appropriate subroutine call. The mapping process is completed by using suitable array names as formal parameters in the subroutine declaration and also in the call to the file reading subroutine REED (see section 8.2). To make this a little clearer, let us consider a hypothetical example where a calling subroutine, ALPHA, calls a subroutine BETA. Within BETA a file of integrals, called AINTS, will be read down into an array A in blocks of 512 words. It is also required, within BETA, to read down two files BMATX and CMATX, of size N^2, into arrays called B and C. The appropriate coding might then be

```
SUBROUTINE ALPHA
REAL ISTORE
COMMON ISTORE (10000)
COMMON/CONSTS/N
.
.
.
II = 513
JJ = II + N*N
CALL BETA(ISTORE(1), ISTORE(II),
ISTORE(JJ))
.
.
.
RETURN
END

SUBROUTINE BETA(A, B, C)
COMMON/CONSTS/N
DIMENSION A(512), B(N,N), C(N,N)
M = N*N
CALL REED (5HAINTS, A, 512)
CALL REED (5HBMATX, B, M)
CALL REED (5HCMATX, C, M)
.
.
.
RETURN
END
```

In this way the arrays A, B and C are, effectively, equivalenced into the appropriate workspace area of the array ISTORE.

7.2. The hash dictionary

All files are known by a five character identifier. A list of all files currently used by OPIT is given in table 1. The bit pattern representing these characters is 'hashed' [23] to yield a target index where (see fig. 2)

$1 < \text{index} < \text{lHDLIM}.$

In the present implementation IHDLlM = 128.

The index produced in this way acts as a pointer to a word in the hash dictionary area of ISTORE immediately above the pointer LOV. The contents of

Table 1
Files names currently used within OPIT

Integral lists	Integral values	Matrictes
KINTS	G-INT	DNSTY
VINTS	K-INT	OVRLP
MINTS	V-INT	KTCNY
	M-INT	HMLTN
		VLUSH
		VCTRH
		EETAA
		VLIST

the word pointed to are arranged so that they, in turn, point into the next area of ISTORE, viz., the file dictionary area. Thus, if ISTORE (index) = 0 in the hash dictionary area, then a new entry is about to be created. A word pair for this new file is created in the file dictionary area using IDPOINT (see fig. 2) to locate the next vacant entry. If ISTORE (index) is non-zero, the entry refers either to a previously created entry for a file of the same name or else, by chance, two hash codes have proved to be identical. To find out which of these alternatives has occurred, a chain structure in the file dictionary (FD) area is followed.

In the subroutine NSEEK, which performs the hashing process, a record is kept of the previous five character name processed. This saves hashing and searching for continuous references to the same file.

7.3. *The file description table*

The limit pointer INDLIM is determined by the number of files to be stored. Two words are allocated for each file entry and the number of files (NFILES) has been set, for the OPIT programs, to 100. A free storage pointer, IDPOINT, points to the next free entry subject to the restriction

IDPOINT $\not>$ INDLIM.

The format of a file dictionary entry in a word pair is shown in fig. 3. It will be seen that the pointer from the hash dictionary (obtained using NSEEK) points to the first word of the pair. The bit pattern

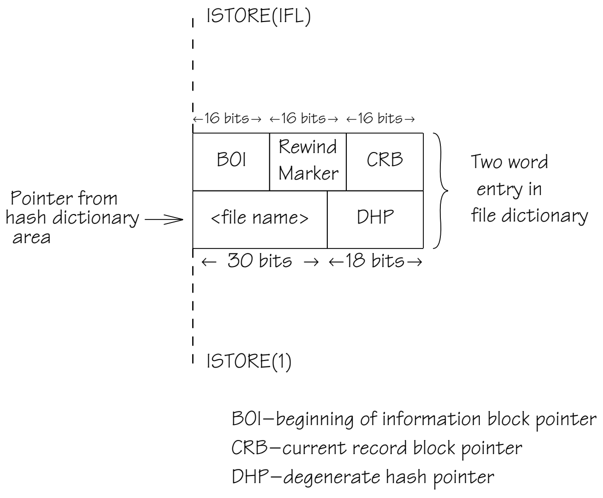
ISTORE(IFL)
←16 bits→ ←16 bits→ ←16 bits→
BOI
Rewind Marker
CRB
Two word entry in file dictionary
Pointer from hash dictionary area
<file name>
DHP
← 30 bits → ←18 bits→
ISTORE(1)
BOI—beginning of information block pointer
CRB—current record block pointer
DHP—degenerate hash pointer

Fig. 3. Format of a two word entry in the file description area of ISTORE.

of the file name currently being processed is exclusive or-ed with (filename) in the FD table entry. If the result is zero, then the correct entry has been found. If non-zero, then a degenerate hash situation has occurred. The degenerate hash pointer (DHP) is consulted, to find the next FD table entry which might match with the name being processed. The exclusive or-ing procedure is repeated at every stage in the DHP-generated chain until a match is found. A setting of DHP = 0 terminates the chain.

When an FD entry has been succesfully located the contents of the second word of the pair can be set to point to entries in the next two areas of ISTORE, viz., the main core pointer area and the disc pointer area. The beginning of information marker (BOI) is set to point to the first record in the main core pointer (MCP) or disc pointer (DP) area that relates to a given file. The current record block marker (CRB) is used when reading from or writing to a given file, and indicates the logical position in the file that has been reached at any given stage of the reading and writing process. The rewind marker is set when a previously created file structure is being overwritten with new information. The action of the BOI and CRB pointers will become apparent when we study exactly how reading and writing is performed.

7.4. Main core pointer and disc pointer areas

Both of these areas consist of single word entries where each word is divided up into three 16 bit fields to be known as the left link, information field and right link respectively.

On initialization the entries in the MCP and DP areas are chained together to form a free storage list. The information fields of successive entries are set to point to blocks of storage either in core (main core pointers), or on disc (disc pointers). Fig. 5a shows the situation in the MCP area on initialization. The set up in the DP area is the same, except that disc addresses are pointed to. When a file is being written entries are claimed from the MCP area (or DP area if core is full) and hence storage of the given type is also claimed. Once MCP or DP entries have been claimed for a given file they correspond to allocated storage and are chained together, in a logical way, for each file that is written (see section 8.1). The two pointers IMCGARB and IDIGARB point to the next free unallocated location in the MCP and DP areas respectively. They are also used to claim new entries from the free storage list when file writing is in process. Their values are initialized to:

$$\mathrm{IMCGARB} = \mathrm{INDLIM} + 1,$$

$$\mathrm{IDIGARB} = \mathrm{IMCLIM} + 1.$$

In order to allocate the space in ISTORE, above LOV, between file dictionary and pointer areas, we shall presume that the limiting factor on a calculation will be the amount of disc store that can be allocated to integral storage.

Let IDBSIZ be the maximum number of words to be transferred to disc in one disc write operation (512 words for this implementation). Let IDBS be the maximum number of entries in the DP list (set to 100 in the current implementation). We find that the total file-structured information that can be stored in the program is:

$$\mathrm{IDBS} * \mathrm{IDBSIZ} + \mathrm{IFL} - \mathrm{IDILIM}.$$

Thus, if we know the amount of core taken by the disc pointer area, and by the area below INDLIM, then we can allocate the remaining core (up to IFL) between main core pointers and main core files. With this in mind we see that IMCLIM is given by:

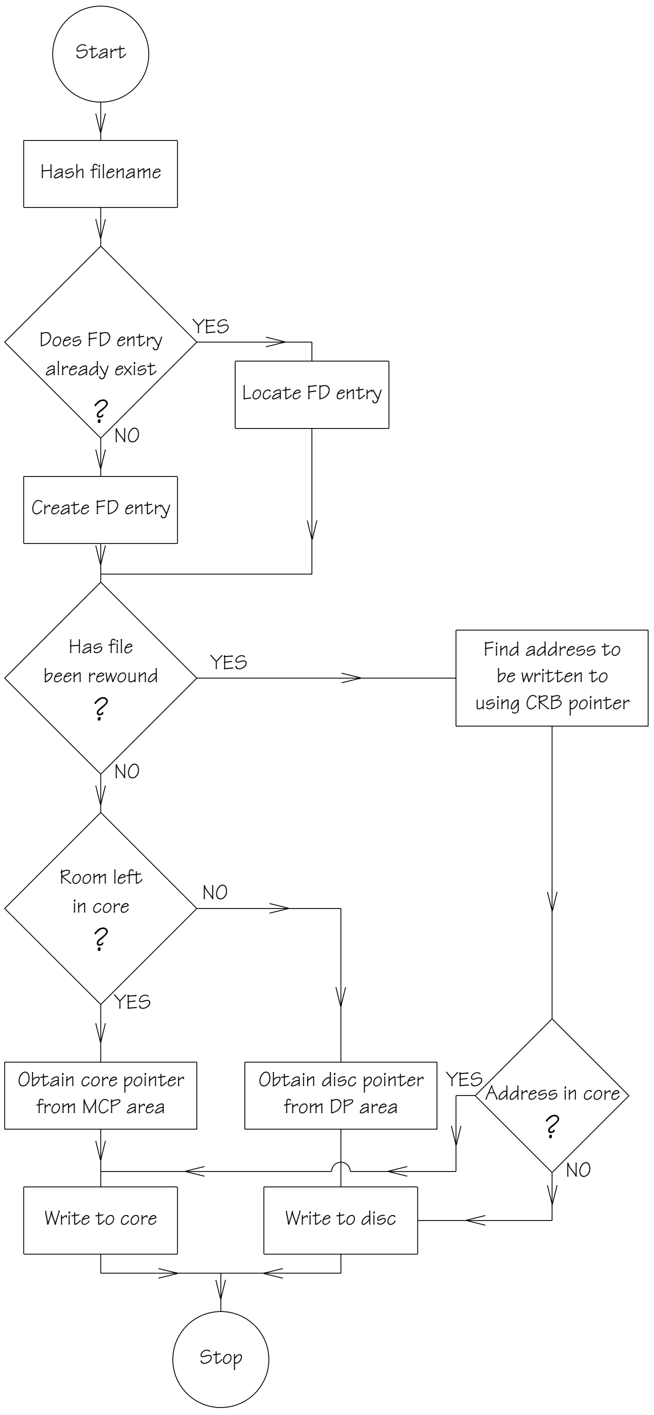

Fig. 4. Flow diagram for file writing.

$$\mathrm{IMCLIM} = \mathrm{INDLIM}$$

$$+ ((\mathrm{IFL} - \mathrm{INDLIM} - \mathrm{IDBS}) / (\mathrm{IMCBSIZ} + 1)) - 1,$$

where IMCBSIZ is the size of a block in the main core storage area.

8. Operations on the file structure

8.1. Writing to a file (subroutine RITE)

In order to exemplify the function of the various areas within ISTORE, the file writing operation will be described in detail. The scheme is illustrated in the flow diagram of fig. 4.

Firstly, the filename is examined to see if an entry for that file already exists. If it exists, the FD entry is examined; if it does not exist an FD entry is created. Next we test to see if the writing operation will over-write previously existing information. This test depends on the setting of a rewind marker in the information field of the FD word pair.

Let us presume that a new file is being written and that there is sufficient space left in core to store the file. A word is obtained from the MCP area using IMC-GARB and IMCGARB is then reset. If this is the first block to be written for the given file then the BOI field in the FD word pair is set. More blocks of core are claimed via the MCP area until sufficient has been allo-cated to hold the file in question.

The structure of the MCP area before and after the write operation is shown in figs. 5a and 5b. Note how the pointers mcb 1 to mcb3 point to starting addresses of in-core blocks which hold the file proper. When the write has been completed, CRB in the FD word pair points to the last MCP word used. Fig. 5b shows that when blocks of storage have been allocated to a file the direction of the left and right pointers for a given chain is reversed compared to the initial set-up (fig. 5a). If a call of the file writing subroutine indicates that there is no space left in core (i.e., IMCGARB > IMCLIM) then an entry from the DP area is claimed, the information field of which will contain a disc address.

If a file is being overwritten (i.e., rewind marker set) then the logical structure of the file already exists within an allocated storage area and it is not necessary to claim free storage words from MCP or DP to add to the list. The CRB will already point to a word containing in its information field the address (in core or on disc) that is to be written to.

8.2. Reading a file (subroutine REED)

This is summarized in the flow diagram of fig. 6. The CRB pointer is first obtained from the FD word pair. The right link of the word pointed to by CRB is copied down to update CRB, and acts as a pointer to the next word to be read.

Next we inspect the information field of a given MCP or DP word. If CRB ≤ IMCLIM the address referred to must be in core. If CRB > IMCLIM the address referred to is on disc. The block of information is then read down from core or disc as appropriate.

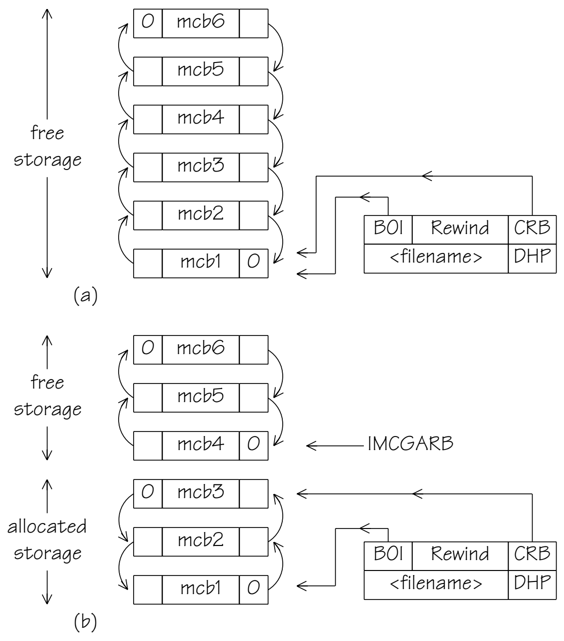

Fig. 5. (a) Part of the main core pointer area just before writing a file. Note how BOI and CRB both point to the bottom of the free storage list. (b) Part of the main core pointer area after writing a file which requires 3 main core blocks. Note how the direction of flow of the pointers is reversed in the allocated storage area compared to the free storage area. Note also the settings of BOI and CRB.

8.3. Logical rewinding of a file (subroutine REVIND)

At the end of a REED or RITE operation the CRB pointer is set to the most recent block accessed. In order to re-read or overwrite a file we first use REVIND to set CRB = BOI in the FD word pair. The information field in the FD pair is set to 0 the first time a file is written. After using REVIND it is set to 1.

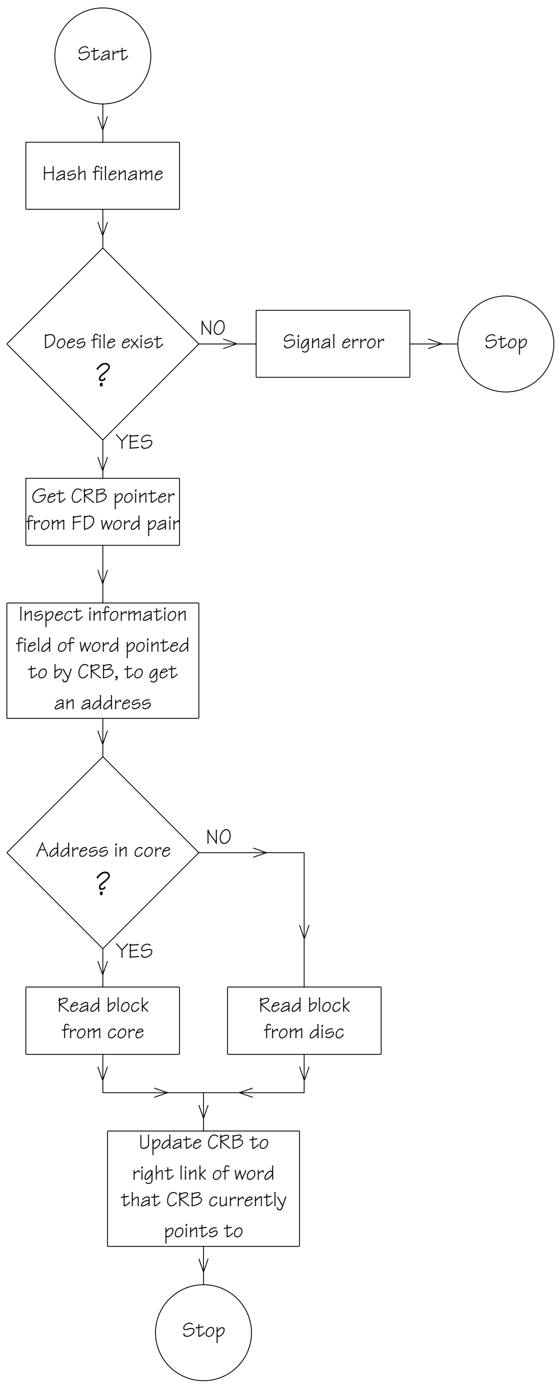
Start
Hash filename
Does file exist ?
NO
Signal error
Stop
YES
Get CRB pointer from FD word pair
Inspect information field of word pointed to by CRB, to get an address
Address in core ?
NO
YES
Read block from core
Read block from disc
Update CRB to right link of word that CRB currently points to
Stop

Fig. 6. Flow diagram for file reading.

8.4. Logical skip backwards (subroutine SKIPB)

This function is required when one wishes to read down a file, block by block, and then to rewrite each block immediately after it has been read.

Study of section 8.2 shows that after a block has been read, CRB is updated so as to access the next block to be read. Thus, if one now performs a RITE operation on the next block is overwritten and not the one that has just been read. Clearly, what is needed is a routine to set CRB to its previous value in the linked list (i.e., the left link of the word that CRB currently points to) and this is performed by SKIPB.

9. Conclusions on effectiveness of the scheme

The file handling scheme works extremely well within the overall framework of the OPIT program as described in sections 2–5. Small problems can now be run completely in core instead of all files being on disc as in the first implementation of OPIT [l]. As an illustration of the advantages of this facility let us consider a calculation on the carbon atom using a basis set of 3 spherical gaussians, whose exponents are to be optimised. In the KDF9 version of the program each ‘function evaluation’ (i.e., each journey round the optimisation loop — see fig. 1) took 4 sec of mill time. In the present version it takes 0.5 sec. These figures do, of course, represent differences in the two programs, other than running the problem totally in core or totally on disc (e.g., FORTRAN code instead of ALGOL). A better test would be to force the 1900 series version of the program to run this small problem on disc, by artificially setting LOV to a very large value (see fig. 2) so that there is room in core only for the file dictionary. This would force all files onto disc storage. Unfortunately there is no way of obtaining a value for how long the machine is occupied in executing the program, i.e., the mill time plus time for disc transfers. The ‘elapsed time’ for the program, in a multi-programming environment, will, of course, include time when the OPIT program is ‘swapped out’ of core onto backing store. Hence, elapsed time is useless for the present comparison.

A second advantage of the present scheme is in its apparent simplicity to the ‘high level user’. That is to say, any person incorporating this file handling scheme into his program will interface with the scheme using only the basic operations of REED, RITE, REVIND, SKIPB. All other manipulations performed by the scheme, such as hashing and dictionary lookup, are invisible to such a user.

The added flexibility of being able to ‘overflow’ from core to disc, automatically, has already been mentioned; it is even possible to have a single file fragmented between core and disc. Perhaps the greatest advantage of the scheme, however, is the fact that almost any number of files can be created, at will, without involving major upheavals in the program coding. We also envisage that such future enhancements as non-serial reading and writing of files, or file deletion and garbage collection [24], can easily be arranged. The scheme could also be extended to permit reallocation of files, e.g., files in core could be transferred to disc and vice versa depending on whether they are going to be heavily used in the next phase of the program. 1t is also worth noting, that if core store is at a premium, there is no reason, in principle, why areas of ISTORE such as the hash dictionary and the pointer chains should not be written out to disc.

The disadvantage of the scheme lies in the initial effort to get it working and the fact that some of the routines have to be written in machine code (see appendix 1). The time overhead incurred by searching in the appropriate areas of ISTORE is more than offset by the advantage of being able to run small problems completely in core, but in order to speed up access of words in ISTORE, all addressing is done relative to the base address ISTORE(0). This base address, which is known after consolidation and loading is stored as a program constant. In order to assist in detecting logical errors in the course of file handling (e.g., an attempt to read beyond the end of a file) a series of traps is built in to the program code, to flag an error in 6 distinct situations. These traps are listed in appendix 2.

One of the principal remaining problems for study, is the optimal setting of the various limit pointers, depending on the nature of the calculation to be performed. For example, two extreme choices for IMCBSIZ (the main core block size) would be N^2 — the size of the smallest file to be created — and $\frac{1}{8} N(N+1)(N^2+N+2)$ which is the largest file created.

If the larger figure is chosen, one can read large files in a single REED operation but storage space will be wasted when storing the smaller files. Conversely, if the lower figure is chosen, storage space is used more efficiently but large files will need several REED operations to access them. Fortunately, the gain in efficiency, especially for small basis sets, has proved so marked, even with non-optimal pointer setting, that the effort in implementing the scheme has been well worth while.

Acknowledgements

The implementation of this 1900 series version of the OPIT program has only been made possible by the unstinted help of other colleagues on the team. We would like to thank Professor G.G. Hall for his constant support for the project from its inception. Thanks are also due to Mr. B. Ford, Mr. J. Prentice, Mr. I. Hume, Dr. A.D. Tait, Miss J. Hylton and all the staff in the Computing Centres of Nottingham and Loughborough Universities.

SRC support for David F. Brailsford during the period of this work is gratefully acknowledged.

Appendix 1

In table 2 are listed the various subroutines inherent in the file handling scheme, together with a brief description of their function. Although ISTORE is a REAL array the bit patterns placed in the various words do not, in general, correspond to floating point numbers. This means, in 1900 Series FORTRAN, that operations such as copying one word to another cannot be performed by standard arithmetic assignment of the type ISTORE(I) = ISTORE(J), since the nature of the bit pattern is checked and an error flagged if it does not correspond to a meaningful floating point number. Thus, in order to make the scheme work on 1900 Series machines, we have had to switch off the overflow register and write the two extra routines COPY and ICHZERO in PLAN. Some of the character handling facilities afforded by the routines in table 2 could also be carried out using standard routines provided by ICL [25].

It should also be noted that the MCP and DP words in ISTORE are divided up into 16 bit fields. Thus all locations within ISTORE must be addressable in 16 bits which limits the overall length of ISTORE to ≈ 65,000 words.

Table 2
Specification of routines needed for the file handling acheme

Name	Subroutine (S) or function (F)	Source language	Description
IHASH	F	PLAN	Returns an index where 1 ≤index ≤128
ICHZERO	F	PLAN	Checks that both half words of a given location are zero
COPY	S	PLAN	Copies contents of one REAL location to another
RAND	S	PLAN	Ands together contents of 2 real locations
REED	S	FORTRAN	Reads a file using OUTLIS
RITE	S	FORTRAN	Writes a file using INLIS
REVIND	S	FORTRAN	Logically rewinds a file (see section 8.3)
SKIPB	S	FORTRAN	Skips back one block in a file (see section 8.4)
INITZ	S	FORTRAN	Initializes the various areas of ISTORE
RXOR	S	PLAN	Exclusive ors together the contents of 2 REAL locations
START	S	PLAN	Finds location of base address ISTORE(0)
ILLSET	S	PLAN	Sets 16 bits of left link
INFOSET	S	PLAN	Sets 16 bits of information field
IRLSET	S	PLAN	Sets 16 bits of right link
ILLGET	F	PLAN	Reads left link
INFOGET	F	PLAN	Reads information field
IRLGET	F	PLAN	Reads right link
SODIT	S	PLAN	Sets DHP in FD word pair
LSBITS	F	PLAN	Reads DHP
NSEEK	S	FORTRAN	Hashes filename and finds FD entry
INLIS	S	FORTRAN	Finds next MCP or DP entry for RITE
OUTLIS	S	FORTRAN	Finds next block to be read for REED

Appendix 2

Table 3 below gives a list of the various error conditions trapped during file handling operations and the name of the routine in which the error is trapped.

Table 3
Error conditions trapped by the file handling acheme

Error Type	Routine where trapped	Nature of error
1	RITE	Overflow of available disc area
2	NSEEK	Attempt to create more than NFILES number of files
3	INLIS	Attempt to write beyond end of file
4	OUTLIS	Attempt to read beyond end of file
5	INITZ	Insufficient core store allocated for running this problem
6	SKIPB	Attempt to skip back on an empty file

References

[1] J .C. Packer, The Nottingham program for closed shell molecular SCF calculations using optimised gaussian orbitals, ed. R.A. Sack. (Copies available on application to DFB).

[2] B. Ford, G.G. Hall and J.C. Packer, Intern. J. Quantum Chem. 4 (1970) 553.

[3] D.F. Brailsford and B. Ford, Chem. Phys. Letters 12 (1972) 60.

[4] A. Veillard, lBMOL IV, Quantum Chemistry Program Exchange, Indiana University, Bloomington, Indiana 47401, USA.

[5] I.G. Csizmadia, J. Moskowitz, M.C. Harrison, S. Seung, B.T. Sutcliffe and M.P. Barnett, POLYATOM, Quantum Chemistry Program Exchange, Indiana University, Bloomington, Indiana 47401, USA.

[6] D.B. Neumann, H. Basch, R.L. Korregay, L.C. Snyder, J. Moskowitz, C. Hornback and P. Leibmann, POLYATOM 2, Quantum Chemistry Program Exchange, Indiana University, Bloomington, Indiana 47401, USA.

[7] M.W. Brinn, Computer Bull. 15 (1971) 316.

[8] G.G. Hall, Proc. Roy. Soc. A205 (1951) 541.

[9] C.C.J. Roothaan, Rev. Mod. Phys. 23 (1951) 69.

[10] D.F. Brailsford and G.G. Hall, Intern. J. Quantum Chem. 5 (1971) 657.

[11] I. Shavitt, Methods Comp. Phys. 2 (1963) 1;

[12] L. Shipman and R. Christoffersen, Computer Phys. Commun. 2 (1971) 201.

[13] R.S. Martin, G. Peters and J.H. Wilkinson, Num. Math. 8 (1966) 203.

[14] H. Bowdler, R.S. Martin, C. Reinsch and J .H. Wilkinson Num. Math. 11 (1968) 293.

[15] P. Dacre, Chem. Phys. Letters 7 (1970) 47.

[16] A.J. Duke, Chem. Phys. Letters 13 (1972) 76.

[17] M.J .D. Powell, Computer J. 7 (1964) 155.

[18] J. Fotheringham, Commun. A.C.M. 4 (1961) 435.

[19] T. Kilburn, D.B.G. Edwards, MJ. Lanigan and F.H. Sumner, I,R.E. Trans.- EC11 2 (1962) 223.

[20] P.J. Denning, Computing Surveys 2 (1970) 153.

[21] FORTRAN 32K Disc Compiler Manual ICL Publication No. 4149.

[22] PLAN Reference Manual ICL Publication No. 4004.

[23] P. Wegner, Programming languages, information structures and machine organisation (McGraw-Hill, New York 1968) ch. 2.

[24] J. McCarthy et aI., LISP 1.5 programmer’s manual (The M.I.T. Press, Cambridge, 1962).

[25] FORTRAN Compiler Libraries Manual, ICL Publication No. 4170.