Discussion and correspondence

A note on the oscillating sort

B. G. T. Lowden

Computing Centre, University of Essex, Wivenhoe Park, Colchester CO4 3SQ

The oscillating sort has been well described in the literature, (Sobel, 1962; Martin, 1971; Knuth, 1975; and Flores, 1969), and differs from most other techniques in that merging and distribution activities are interspersed.

If the initial number of unit strings is S then the number of times each string is passed, i.e., the number of merges will be $[\log_{T-2} S]$, where $[x]$ = the smallest integer $\geqslant x$, and the total volume passed during the sort will be $S[\log_{T-2} S]$ where T is the total number of drives available including input.

Like most other sort merge techniques the value of S is critical since a small increase over a power of $T - 2$ necessitates an additional merge pass.

The upper boundaries of these merge points may, however, be raised by including an additional string from core at each successive merge, a modification mentioned in Sobel (1962) but, to the author's knowledge, not pursued further.

The result is an increase in the number of strings which may be sorted with n merges from $(T-2)^n$ to

$$(T-2)^n + (T-2)^{n-1} \ldots (T-2) + 1 \equiv \frac{(T-2)^{n+1} - 1}{T-3} \; ;$$

for values of S lying between these limits a complete merge pass is thus saved, over the standard sort, and volume passed is reduced by a factor of $n/(n+1)$. This improvement is shown diagrammatically in **Fig. 1** for the case of $T = 5$.

Merging performance is, however, not only dependent on volume passed since block size will also affect the total tape read/write time.

Consider a standard oscillating merge problem with buffer size N which utilises T tape drives. Assuming that the block size of each tape is the same and since $T - 1$ tapes require buffer space during a merge, we may allocate a maximum block size to each tape of $N/(T-1)$. The modification described above, however, implies that all T tapes require buffer space simultaneously, leading to a block size of N/T. Clearly this increases the read/write time by a factor of

$$\rho = \frac{N + TG}{N + (T-1)G} \text{ where } G = \text{interblock gap size}$$

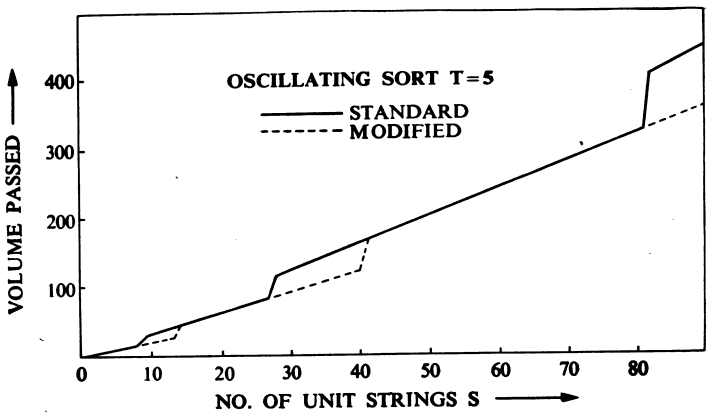

Fig. 1

and this must be offset against the reduction in volume passed. Provided therefore $\rho.n/(n+1) < 1$ the modification leads to improved performance and is particularly advantageous for small T and large N, which is a common characteristic of small single-programmed machines.

An example

Let:

No. of tape drives available	$T = 5$
Allocated buffer space	$N = 40$K bytes
File size	$S = 3{,}000$ records
Interblock gap size	$G = 0{\cdot}75$ in.
Packing density	$P = 800$ b.p.i.

Then:

$$\rho = \frac{N + TG}{N + (T-1)G} = 1{\cdot}0001 \; n/(n+1) = 7/8 = 0{\cdot}875$$

and sort time is reduced by 12·4 per cent using the modified technique.

References

SOBEL, S. (1962). Oscillating sort—a new merging technique, *JACM*.

MARTIN, W. A. (1971). Sorting, *Computing Surveys, ACM*.

KNUTH, D. E. (1975). *The art of computer programming*, Vol. 3, Addison Wesley.

FLORES, I. (1969). *Computer Sorting*, Prentice-Hall.

To the Editor
The Computer Journal

Sir

The recent paper by Burkhard (*The Computer Journal*, Vol. 18, August, 1975, pp. 227-330) purports to give an algorithm for non-recursive tree traversal which uses a fixed amount of storage and no additional flag bits in the nodes. While this contention is literally true, Burkhard places a restriction on the field values which is equivalent to providing a flag bit for each: 'For every node N in T with address P = P(N) LPNTR(P) > P and RPNTR(P) > P.' The algorithm which he states for performing the traversal is identical to the well-known 'pointer reversal' method described by Schorr and Waite [*CACM*, Vol. 10, pp. 501-506]. This algorithm is also discussed by Knuth in one of the references given by Burkhard (*The Art of Computer Programming*, Volume 1/Fundamental Algorithms, pp. 417-419).

The minimum information required by a non-recursive tree traversal which does not permanently alter the data structure seems to be three addresses plus one bit per level. Burkhard's algorithm does not meet this bound. The restriction is applied to *both* fields of every element, thus providing two bits of information per element. Schorr and Waite use one bit per element, less than Burkhard but still more than the minimum.

Burkhard's algorithm thus provides us only with an extra trick for encoding the necessary information. Such tricks are useful, and should be disseminated, but they should be more carefully placed in proper context.

Yours faithfully,

W. M. WAITE

Department of Electrical Engineering
University of Colorado
Boulder
Colorado 80309
USA
13 February 1976

To the Editor
The Computer Journal

Sir

Natural programming

In the May 1976 issue of *The Computer Journal*, J. Inglis appealed for inclusion of a system boolean end-of-file function in programming languages, claiming that 'perhaps the most significant contribution would be that *beginners* could write natural well-structured programs . . .' Of his paradigm he asserts that 'It is unfortunate that this type of program cannot usually be expressed naturally in current programming languages and that students are compelled, very early in their learning, to resort to an unnatural approach which sows the seeds of bad programming style.'

The importance thus given to naturalness in programming is most laudable, and the importance is for all who program—not just the beginners. However, Inglis' examples and his paradigm display an unnaturalness of much more frequent significance. Phrases such as **while not** and **if not** are usually quite unnatural compared to the words **until** and **unless** respectively, as their substitution in Inglis' examples will show. Programs using such words would be easier to produce and understand.

Yours faithfully,

W. N. HOLMES

IBM Australia Limited
80 Northbourne Avenue
Canberra
A.C.T. 2601
Australia

Reference

INGLIS, J. (1976). Structured programming and input statements. *The Computer Journal*, Vol. 19, No. 2, pp. 188-189.

To the Editor
The Computer Journal

Sir

Self-confidence in a computer

The germ of the ideas introduced here arose out of the question whether a computer could be built which could gradually develop a doubt whether it was on the right track—this being a feature of natural intelligence that the present generation of computers does *not* show.

After considerable abortive casting around, the first real breakthrough came with the realisation that in Triple Modular Redundancy (TMR), where every module is triplicated and the results passed through a majority vote taker, we have the following situation, when every module has a chance p of functioning correctly and $q = 1 - p$ of being in error,

	probability	*result*	*suspicion*	*observed frequency*
All correct	p^3	correct	No	M
One wrong	$3p^2q$	correct	Yes	$N(+E)$
Two wrong	$3pq^2$	incorrect	Yes	(E)
All wrong	q^3	incorrect	No	negligible

We have to assume that q is sufficiently small that the last possibility can be neglected. In this case we have the observed frequencies M of correct results and N of suspicious ones, from which we can deduce the expectation $E = N^2/3M$ of an actual error getting through. If $E \ll 1$ then we can be qualitatively confident in the result, but assuming a Poisson distribution we can go further and state that $e^{-E} \simeq 1 - E$ is the probability that our results are correct. There would seem to be no reason why a system built in this way should not invariably attach this figure to any results it prints.

TMR is very expensive. In a multiprocessor environment, two results that agree need not waste time on a third, but if we exploit this, then we have $E = 3N^2/4M$. In a different context, if a check on memory is kept by an error correcting code rather than by a single parity bit, and if it is assumed that single bit errors occur at random, then similar arguments lead to an expectation $E = N^2/M$.

So long as the probabilities of error computed in any of these ways from different sources are small, they can be combined by addition, so that the system overheads need not be high.

These estimates of the absence of error relate to hardware faults. The common principle behind them is that by employing an error *correcting* system and keeping a record of both error-free and error-corrected applications, an estimate can be obtained of the chance of an error slipping through the correction system. The exact formula depends on the technique used in the process of correction. Recently (James and Partridge, 1976), error correcting techniques have been applied to spelling mistakes and syntax errors, and, since doubt about the final reliability of a result can come from any level from hardware right up to the logic of a program, one wonders whether the principle can be extended to cover these levels.

Several difficulties have already become apparent, and there is no automatic route to an estimate-formula. Consistent mis-spelling of one reserved word is very different in its relation to doubt from general carelessness, for example. The ratio of initial to return entries to a loop can be monitored but only the programmer can tell, at present, what the reasonable limits of this ratio are in a given case.

Ironically, small changes in hardware design would be necessary to provide automatic provision of hardware reliability estimates, but software experiments need no such precondition. The advantages of free format languages are now generally realised, but this would not prevent use of format evidence in checking correctness. We look forward to the day when a computer will accept and execute any syntactically and logically correct program, but will express considerably less confidence in the result if it is submitted in a slovenly layout!

Yours faithfully,

M. ALAUDDIN and B. HIGMAN

Department of Computer Studies
University of Lancaster
Bailrigg
Lancaster

Reference

JAMES and PARTRIDGE (1976). *The Computer Journal*, Vol. 19, No. 3, pp. 207-212, and references given there.

To the Editor
The Computer Journal

Sir

I notice that *The Computer Journal* does not impose a uniform style for references given at the end of articles: sometimes only the first page number is given, sometimes first and last. Might I suggest that the 'first and last' style be made obligatory. Its main advantage is, of course, that it allows one to specify precisely the pages to be Xeroxed when writing to libraries holding bound copies of the journal in question. Also, as you would probably agree, it is often useful to know whether a given title refers to a one-page note or to a major twenty-page paper.

Incidentally—and unrelatedly—minimal editorial courage would quietly replace 're-cap' by 'recapitulate' on page 192 of the last issue. One must draw the line somewhere.

Yours faithfully,

G. A. ERSKINE

Data Handling Division
CERN
1211 Geneva 23
Switzerland
24 June 1976

Editor's note:
It is the deliberate policy of the *Journal* not to restrict unreasonably the freedom of an author to choose his own style; in any case it is

not practicable for the amateur staff of the *Journal* to undertake the workload involved in extensive editing. Nevertheless the suggestion of Dr. Erskine regarding references clearly is valuable. Would authors please take note.

To the Editor
The Computer Journal

Sir

Structured programming and input statements

Inglis's Paper (this *Journal*, Vol. 19, No. 2, May 1976) leaves some major difficulties unanswered, and I think they must be introduced into the discussion. The difficulty is not, as I think he is suggesting, simply that the input statements don't tell the program that end-of-input is coming next—they don't tell it what is coming next *at all.* If the program is designed with a structure which separates out the various parts of the processing, then how do we arrange for the machine to be in the correct part of the program if we don't know what data is coming next. I am, of course, assuming that, in the context of this discussion, the reader understands clearly that what processing has to be performed next depends on what data is delivered by the input statements. In the example given by Inglis, if end-of-file turns up then the 'finish' process has to be performed next.

The general difficulty is that there are different types of records, and/or groups of records, which are recognised by values within the records themselves, rather than by values preceding the records. To be consistent, the Inglis approach would have to attach all these values to the preceding record. Consider the development of the following example (using Inglis's informal ALGOL-like notation).

Imagine a program which processes a file containing two records, a Type 1 followed by a Type 2, identified by values within the records themselves. Imagine, further, that no errors are possible, so the file really is two records, a Type 1 followed by a Type 2, and nothing else. The program will look like this:

Initialise;
do [*read*(*file*); *Process.type*1];
do [*read*(*file*); *Process.type*2];
finish;

Observe that the program knows what is coming next. It has no need to test the record type, or guard against end-of-file, given the specification above.

Imagine, now, that we specify any number of Type 1's, followed by any number of Type 2's, and that 'any number' includes zero. Again excluding errors, the program will now be

initialise;
read(*file*);
while *type*1 **and not** *end.of.file*
do [*process.type*1; *read*(*file*)];
while *type*2 **and not** *end.of.file*
do [*process.type*2; *read*(*file*)];
finish;

where the *read* has the traditional definition of setting *end.of.file* to **true** after the first unsuccessful read.

Changing the *read* to the form suggested by Inglis (the system boolean function *end.of.file* becoming **true** in concert with the appearance of the last record) we would have

initialise;
while *type*1 **and not** *end.of.file*
do [*read*(*file*); *process.type*1];
while *type*2 **and not** *end.of.file*
do [*read*(*file*); *process.type*2];
finish;

The problem now is that although the system is kind enough to tell the program that the next *read* will produce a record (or not), its benevolence does not extend to telling the program which type it is going to be. Suppose there are zero Type 1 records?

We could, of course, change the program structure to

initialise;
while not *end.of.file*
do [*read*(*file*); *process.record*];
finish;
process.record: **if** *type*1
then *process.type*1;
else *process.type*2;

(remember that we excluded errors, so if it's not a type 1, it must be a type 2). Now we have two problems:

there may be no records at all on the file. This is not difficult to surmount: we merely require to treat the absence of a first record as a special case, with *end.of.file* being capable of yielding **true** before any *reads* have been performed.

the program is now processing a file which is an iteration of records each of which can be either a type 1 or a type 2. This is much more serious. Not to put too fine a point on it, the structure of the program is now WRONG.

This second problem comes to the heart of the matter: it is almost invariably wrong to see a file as a simple iteration of records. Usually they are grouped together in some way which is either 'natural', or necessary, to the problem being processed. The user asking for a modification which he considers to be quite simple, but which proves very difficult to implement is a common symptom of failing to see these groupings.

Consider the following request for a modification:

'the program is a big success, and now has far more users than we ever expected. What we now want is to run through any number of record groups each of which consists of a whole lot of type 1's, followed by a whole lot of type 2's. The user will be identified on a type 3 record at the start of each record group. Since the operators will be inserting the type 3 records, it is possible, regrettably, that they might occasionally be missing. If there is no type 3 present at the start of a group, identify that user by the date and time of day.'

How are we going to implement this?

We will have to change the program to

Initialise.first.group;
while not *end.of.file*
do [*read*(*file*); *process.record*];
finish.last.group;
process.record: **if** *type*1
then *process.type*1;
else if *type*2
then *process.type*2;
else [*finish.group*; *start.group*];

This is the cheap solution—if the type 3 is omitted, its group is concatenated with the previous one (unless it's the first one!). Additionally, the first group is always initialised, and the last one always finished, even when there are no groups at all. We will have to solve these problems either by appealing to the user, pointing out how unlikely they are to occur, or by setting and testing switches—they are the only answers now.

The sad thing is that what we are trying to cure are all self-inflicted wounds—the file is not a simple iteration of records. Unless Inglis proposes that there be system functions which allow the program to interrogate all recognition values of the following record, then the programmer will be invited to fall into this awful trap.

Inglis states that it is difficult to find more than 'passing references' to input statements in the literature of structured programming. Jackson (1976) deals in some detail with a method which is consistent. If we obey the following simple rules, the program will be able to see all the necessary values of the next record:

1. Read a record at the very beginning of the part of the program which processes the whole file (i.e. immediately following the OPEN).
2. Always view the tests as testing the 'next' record.
3. Always view the process.record level as processing a 'current' record.
4. Always read, to replace the 'next' record, immediately at the end of processing the 'current' record, thus ensuring that, except when actually processing a record, the next one is always available for inspection.
5. Form the read process such that it treats the end.of.file condition in a way consistent with the other recognition values. This in practice means a simple 2-valued flag—'on' for *end.of.file*, 'off' for **not** *end.of.file*.

The program then has the structure

initialise;
read(*file*);
while not *end.of.file*
process.group;
finish;
process.group: *initialise.group*;
if *type*3

then do [*process.type3*; *read(file)*];
else *process.date.and.time*;
while *type1* **and not** *end.of.file*
do [*process.type1*; *read(file)*];
while *type2* **and not** *end.of.file*
do [*process.type2*; *read(file)*];
finish.group;

which is much more 'natural'.

Yours faithfully,

M. V. SLAVIN

Michael Jackson Systems Limited
101 Hamilton Terrace
London NW8 9QX
9 July 1976

Reference

JACKSON, M. A. (1976). *Principles of Program|Design*. Academic Press.

To the Editor
The Computer Journal

Sir

The May 1976 issue of *The Computer Journal* included a short paper by J. Inglis, discussing structured programming and input statements.

To write goto-free programs with input statements without the proposed end.of.file system function, one could define the input statement like:

reading.ok (*x*): 'a boolean function: if end of file *x* has been reached then return false; otherwise, make the next record from file *x* available to the program and return true'.

A program using this function could look like:

initialise;
while reading.ok (*x*) **do**
process.record;
finish;

Yours faithfully,

T. S. MONSEN

A/S Computas
Veritasveien 1
Oslo
Norway
28 July 1976

Mr. Inglis replies:

Mr. Monsen proposes the use of a boolean procedure with a side-effect in order to achieve a **goto**-free program for the class of problem I discussed. This proposal is of course applicable only to programming languages which allow such a procedure to be defined; moreover, the use of such a procedure leads to program opacity and to errors, and I would hesitate to introduce an example to beginners. Quite apart from these considerations, I retain my belief that it should be possible to test for end-of-file without attempting to read a record, just as it is possible to test for a zero divisor without attempting to divide. A similar view was expressed as long ago as 1961 by a working committee of BCS Group 5 (see Willey *et al.* (1961)).

Mr. Slavin is perhaps right in widening the scope of the discussion to include other classes of problem; such matters certainly require an airing in an academic community which claims to be concerned with the structure of programs. But his view of my favoured end-of-file function as 'telling the program that end-of-input is *coming next*' is a consequence of his acceptance of the limitations of the currently popular languages. The end-of-input condition *already exists* as soon as the last record of the file has been delivered to the program; it is often quite wrong structurally to detect it by the failure of a further *read* statement. This is not to say that current facilities should be abolished, but simply that they are limited.

However, let us take Mr. Slavin's view that we are considering two alternatives and that, for a sequentially accessed file, a programming language may provide only one of the following end-of-input functions. The function **end.of.input**(*x*) may be defined:

1. (Inglis) It yields the value **true** whenever no further records exist in file *x*; otherwise, it yields the value **false.**
2. (Slavin) It yields the value **true** at any time after a **read** statement for file *x* has been executed when no further records exist in file *x*; otherwise, it yields the value **false.**

Now consider two classes of program:

A, in which the file is seen as a simple iteration of records;
B, which requires a record look-ahead capability, as in Mr. Slavin's example.

Function 1 permits the natural expression of class A programs, as shown in my paper. It also permits the natural expression of class B programs (and this is the point Mr. Slavin has missed). Whatever objections a purist might have to the boolean expressions in Mr. Slavin's final program, it can be regarded as a well-structured program of class B. In that program, the variable *end.of.file* may be taken as user-defined, like *type1*, *type2* and *type3* (and, like them, requiring initialisation at the time of opening the file), and the user's procedure *read*(*file*) may be defined as:

if end.of.input(*file*) **then** *end.of.file* := **true**
else read(*file*);

Function 2 permits the natural expression of class B programs—simply substitute **end.of.input**(*file*) for each occurrence of *end.of.file* in Mr. Slavin's final program—but, as I showed, it does not permit the natural expression of class A programs. The practical hints quoted from Jackson's book (which, incidentally, was published after my paper was in the hands of the Editor) are an example of what one should *not* have to do when, for example, one simply wants to count the number of records in a file. Despite Mr. Slavin's assertion to the contrary, programs of class A are not so uncommon, especially among casual computer users. Mr. Slavin is in the curious position of favouring a language feature which is suitable for more complex programs, but which makes the natural expression of many simple programs impossible.

To state the situation more concisely, function 1 may be used to provide function 2, but function 2 cannot be used to provide function 1. Both functions are desirable. Function 1 should therefore be the choice of the language designer.

Reference

WILLEY, E. L. *et al.* (1961). *A critical discussion of COBOL*. *Annual Review in Automatic Programming*, Vol. 2, Pergamon Press, pp. 293-304, at p. 298.

To the Editor
The Computer Journal

Sir

Mr. H. A. Marriott has written in the May 1974 issue of your *Journal* that he has some apprehensions about the spelling of ALGORITHM and he has further cited the Oxford Dictionary which quotes ALGORISM instead, which is derived from the name of an Arab Mathematician and further states that the -ITHM ending is a mis-spelling.

In this connection we would like to state that the word ALGORITHM has a peculiar linguistic background. The said word consists of the two words viz. 'Al' and 'Gorithm'.

'Al' means the set of all alphabates (consonants and vowels) in Pāṇini's (Sanskrit) linguistics. Pāṇini, the ancient Indian Grammarian has divided this set of alphabates into 14 primary subsets with end points.

These primary subsets and their definitions in BNF notations run as follows.

1. A I U (ṇ)	⟨A ṇ⟩ := A | I | U
2. Ṛ Ḷ (k)	⟨Ṛ k⟩ := Ṛ | Ḷ
3. E O (ṅ)	⟨E ṅ⟩ := E | O
4. AI AU (ç)	⟨AI ç⟩ := AI | AU
5. H Y V R (ṭ)	⟨H ṭ⟩ := H | Y | V | R
6. L (ṇ)	⟨L ṇ⟩ := L
7. Ñ Ṁ Ṅ Ṇ N (ṁ)	⟨N ṁ⟩ := N | Ṁ | Ṅ | Ṇ | N
8. JH BH (ñ)	⟨JH ñ⟩ := JH | BH
9. GH ḌH DH (ṣ)	⟨GH ṣ⟩ := GH | ḌH | DH
10. J B G Ḍ D (ś)	⟨J ś⟩ := J | B | G | Ḍ | D
11. KH PH ÇH ṬH TH (v)	(KH v) := KH|PH|ÇH|ṬH|TH
12. Ç Ṭ T K P (y)	⟨Ç y⟩ := Ç | Ṭ | T | K | P
13. Ś Ṣ S (r)	⟨Ś r⟩ := Ś | Ṣ | S
14. Ḥ (l)	⟨Ḥ l⟩ := Ḥ

The contiguous primary subsets can be added together to form the hyper subsets.

$$\langle A\ \dot{c}\rangle := (A\ \dot{n}\rangle \mid \langle \dot{R}\ k\rangle \mid \langle E\ \dot{n}\rangle \mid \langle AI\ \dot{c}\rangle$$

'Ac' (Pronunciation 'Ach') means union of the first four primary subsets (1, 2, 3 and 4). In other words it is union of the primary subsets from a set (1) whose first symbol is 'A' through a set (4) whose end point is ç 'A ç' means a set of all vowels of the sanskrit language. 'Hl' means union of primary subsets from (5) through (14) and it means the set of all consonants in the sanskrit. Finally 'A l' is the union of all the 14 primary subsets and means the set of all vowels and consonants in the sanskrit.

By the passage of time, 'Al' was used in the sense of All dropping the sense of consonants and vowels. Even the linguistic similarity between the words 'Al' and All is remarkable.

'Gorithm' the second part, is derived from the Sanskrit stem root 'Granth' meaning 'to string together, to compose, to fasten, to put together'. The linguistic variations from GRANTH to GORITH are easily understandable. The connotation of the word ALGORITHM also substantiates perfectly its scientific linguistic basis.

On this background, the explanation offered by the Oxford Dictionary sounds far from convincing.

Yours faithfully,

S. N. BALDOTA and V. K. KSHIRSAGAR*
Electronic Data Processing Centre
University of Bombay
Bombay 400 020
India

*Elphinstone College
Bombay 400 032
India
26 August 1976

To the Editor
The Computer Journal

Sir

Fast input/output of variable-length arrays in FORTRAN IV

In the *Journal* of August 1976, R. Taylor describes a valuable method of increasing the efficiency of FORTRAN programs produced by some compilers.

It is unfortunate that he suggests that this method 'ought to work with any FORTRAN compiler'. At least one compiler rejects such statements in accordance with section 7.2.1.1.2 of the FORTRAN Standard (ISO, 1972) which states 'The values of the actual arguments that represent array dimensions in the argument list of the reference must be defined prior to calling the subprogram and may not be redefined or undefined during execution of the subprogram.'

There are still good reasons for using non-standard FORTRAN in some situations, but any publication which does not conform to the Standard should state this explicitly.

Yours faithfully,

D. HITCHIN
Research Support Unit
School of Social Sciences
The University of Sussex
Falmer
Brighton BN1 9QN
2 September 1976

Reference

International Organisation for Standardisation (1972). ISO Recommendation R1539, 'Programming Language FORTRAN'.

Dr. Taylor replies;

Mr. Hitchin is right. However my suggestion at least concentrates the non-standard code. If the data were suitably written, one could observe standards by reading N in the calling, and the partial array in the called, routines. The important point is to try always to transfer continuous blocks of data and avoid looping through subscripted elements.

To the Editor
The Computer Journal

Sir

Symposium on the use of computers in shipboard automation

May I draw your readers' attention to one or two errors of fact and interpretation in the review of the above symposium published in the August 1976 issue of your *Journal*? Firstly, the purpose of the study on which the symposium was based was to explore the *potential* for ship automation systems based on onboard computing facilities and not to survey the progress of ship automation generally, as stated by your reviewer. Secondly, the statement that there was little contribution from equipment manufacturers is incorrect in that U.K. manufacturers were involved in the direction of the work and in fact were joint sponsors. During the course of the study discussions were held with all major U.K. and Scandinavian equipment manufacturers.

The economic case for bridge automation systems derived mainly from predicted reductions in fuel consumption and voyage time rather than enhanced safety. Safety benefits, although important, were assessed conservatively in view of the uncertainties in the data and were credited with a relatively small contribution. Your reviewer's comment on the use of digital techniques for machinery control overlooks the conclusions drawn on potential cost/reliability benefits, and omits mention of operational benefits (difficult to quantify economically at this stage) in the important areas of machinery surveillance and condition monitoring.

Finally, it is fair to point out that in compressing the large amount of material in the original study report into a series of papers of reasonable length, much detailed information had to be omitted, which may account for your reviewer's final comment. Nevertheless it was considered that in summarising the results of what was essentially a feasibility study emphasis should be given to discussion of the many factors influencing viability and future trends, and that an authoritative statement of these matters would be helpful to many whose experience has been only in one or other of the three industries concerned. It is interesting to note that while no U.K. national project has yet been undertaken to promote development in this field, commercial developments since the study was completed in 1973 have generally been following the broad pattern envisaged in the report.

Yours faithfully,

H. C. WILKINSON
British Ship Research Association
Wallsend Research Station
Wallsend
Tyne and Wear
NE28 6UY
10 September 1976

Erratum

Formula (3) of the paper 'Hit ratios' by S. J. Waters (this *Journal*, Volume 19, No. 1, February 1976) should read:

$$BHR = 1 - \frac{{}_{N-B}C_H}{{}_{N}C_H} \qquad (3)$$

Similarly in Appendix 3.