Source language debugging on a small computer

R. H. Pierce

*Ferranti Limited, Bridge House, Gatley, Cheshire**

Conventional dynamic debugging techniques are often restricted to machine level programs, while fully conversational compilers for powerful languages are costly to develop and will not fit into small systems.

This paper describes a source-language debugging system for CORAL 66 which allows a user to run his CORAL program under teletype control, and to make changes to the source program. The changes are compiled by the debugging program and immediately incorporated into the object code. This provides a useful facility which can be added to all but the smallest systems.

(Received May 1973)

1. Introduction

It is now common practice to use a high-level language for development of both systems and applications software, even on small computers. However, it is unfortunately true that while compilers abound, the same cannot be said of good run-time diagnostic and debugging aids. On a machine with a multiplicity of order formats and addressing schemes such as the Argus 700 (Eyre & Williams, 1973), a user would require a considerable knowledge of both the machine and the compiler to debug a high-level language program at the level of the machine (e.g. from a core dump). Now the intention is that all programming for the Argus 700 should be carried out in CORAL 66, FORTRAN or some other high-level language, and that the machine code should be of concern only to the compilers. It is thus reasonable that the programmer be given efficient debugging tools that require only a knowledge of the source language, and do not presuppose any familiarity with the machine (apart from knowing the word length). Tools of this nature must inevitably reduce the cost of programming (and hopefully increase user satisfaction).

Two such systems for CORAL programs are supported. The first consists of a fairly conventional trace package, with tracing on a printer controlled by compile-time directives and run-time steering commands. The second is a more sophisticated dynamic debugging facility, which allows the user to control the execution of his CORAL program from a teletype. The point about this system is that it works entirely in terms of the source program. As far as the user is concerned, the object program is invisible. This paper describes the Argus 700 program called Dynamic Debugging-Source (DDS).

2. The DDS system

The Argus 700 is a small to medium scale machine, and the size of most configurations would preclude the use of a full scale interactive compiler such as IBM's PL/I Checkout Compiler (Cuff, 1972). In addition to this, the development of the main three-pass CORAL compiler was well under way when the need for an interactive debugging system was recognised, and consequently it was decided to construct a separate source-language debugging system that would fit into a minimum amount of space while allowing the user to make minor alterations to his CORAL program without having to recompile the program (such alterations are commonly known as 'patches'). In some respects, DDS is similar to a small-scale interactive compiler, but the original compilation is carried out by the full CORAL compiler. DDS operates in conjunction with this compiler, though they are separate programs.

The compiler produces a map of the source program containing all the information that the debug needs to run the program. This map is held on disc. It will only be written if specifically requested by a steering command to the compiler.

In addition the compiler will provide the user with a listing of the source program in which each executable statement is assigned a serial number (**Fig. 1**). The user refers to this numbered listing when debugging his program.

When using DDS, a programmer at a keyboard may:

1. Set (or clear) a number of breakpoints in his program.
2. Resume execution of the program at a specified source statement, or continue from a breakpoint.
3. Examine and alter variables, array elements or absolute locations.
4. Delete, alter or insert CORAL statements at any point in the program, or remove a patch previously inserted.

Commands may be entered whenever a program stops at a breakpoint, or is interrupted by the user's typing a control character on the keyboard. Run-time errors also cause control to be passed to the user, after the appropriate message has been output.

Commands are as abbreviated as possible; lengthy command formats would tend to irritate most users. If a mistake is made when typing a line (either a command or a CORAL statement) the character? will cause the entire line to be ignored.

The general format for commands is:

```
⟨command⟩ ::= ⟨store examination command⟩|
              ⟨patching command⟩|
              ⟨breakpoint command⟩|⟨other
              command⟩
⟨store examination command⟩ ::= ⟨general command⟩
⟨breakpoint command⟩ ::= ⟨general command⟩
⟨other command⟩ ::= ⟨general command⟩
⟨patching command⟩ ::= X ⟨from st no⟩⟨to st no⟩
                         ⟨newline⟩
                         ⟨coral patch⟩|
                         Y ⟨statement number⟩⟨newline⟩
                         ⟨coral patch⟩|
                         R ⟨statement number⟩
⟨statement number⟩ ::= ⟨decimal number⟩
⟨from st no⟩ ::= ⟨decimal number⟩
⟨to st no⟩ ::= ⟨decimal number⟩|⟨nil⟩
⟨coral patch⟩ ::= ⟨ss list⟩⟨newline⟩|⟨newline⟩
⟨ss list⟩ ::= ⟨subset statement⟩|
              ⟨ss list⟩⟨newline⟩⟨subset statement⟩
⟨general command⟩ ::= ⟨command identifier⟩⟨space⟩
                       ⟨parameters⟩⟨newline⟩
⟨parameters⟩ ::= ⟨parameters⟩⟨space⟩⟨parameter⟩|
                 ⟨parameter⟩
```

*Now with ICL, Wenlock Way, West Gorton, Manchester M12 5DR.

```
[0064] CBST:
           ALPHA:=ALPHA+2;
[0065]     'IF' SYNSYM 'LT' 128 'THEN'
[0066]        'BEGIN'
[0067]          SALINK;
[0068]          TAKES;
[0069]          ALPHA:=ALPHA;
              'END' 'ELSE'
[0070]        'BEGIN'
[0071]        'IF' SYNSYM 'GE' 256 'THEN'
[0072]             'BEGIN'
[0073]               BOOL:=0;
[0074]               SEGCALL;
                    'END' 'ELSE'
[0075] [0076]        'IF' SYNSYM 'LT' 240 'THEN' 'GOTO' CASE1 'ELSE'
[0077] [0078]        'IF' SYNSYM 'LT' #372 'THEN' 'GOTO' CASE2 'ELSE'
[0079] [0080]        'IF' SYNSYM 'EQ' CNXT 'THEN' BOOL:=1
[0081]               'ELSE' BOOL:=0;
[0082]       CEXIT(BOOL);
               'END';
[0083]     'GOTO' NXTSYM;
[0084] CERR:
           ERREC(ALPHA);
[0085]     'GOTO' NXTSYM;
```

Fig. 1 Numbered source listing

⟨parameter⟩ ::= ⟨number⟩|⟨identifier parameter⟩
⟨identifier parameter⟩ ::= ⟨identifier⟩⟨scope⟩|
⟨identifier⟩[⟨subscript⟩]
⟨scope⟩
⟨scope⟩ ::= : ⟨decimal number⟩|⟨nil⟩
⟨subscript⟩ ::= ⟨number⟩
⟨number⟩ ::= ⟨sign⟩⟨decimal number⟩|⟨sign⟩
⟨decimal number⟩.
⟨decimal number⟩| #⟨octal digits⟩|
@⟨hexadecimal digits⟩
⟨sign⟩ ::= −|⟨nil⟩
⟨space⟩ ::= at least one space
⟨newline⟩ ::= at least one new line (CR LF).

The syntax of ⟨subset statement⟩ is given below.

The command identifier is in most cases a single letter. **Table 1** gives a list of the available commands. Identifiers may be scalars or arrays of any arithmetic type, and must be declared somewhere in the program. Since CORAL is a block-structured language, the same identifier may be declared a number of times, so clearly it is necessary to provide a means of specifying which declaration is meant in any given command. The form ⟨identifier⟩:⟨number⟩ is used for this purpose, the ⟨number⟩ being the statement number attached to the 'BEGIN' of the block containing the required declaration. The default option is the block containing the breakpoint or fault at which the program stopped. All arrays are regarded as being one dimensional for the purposes of debugging, with lower bound equal to that of the first bound-pair of the array. Thus, to examine or alter an element of a multi-dimensional array, it is necessary to regard the array as a vector and work out the correct subscript. In CORAL, arrays are stored in row-major order. Identifiers as parameters are only allowed in the store examination and alteration commands.

Printing is done by the debugging program in a style that depends on the type of the identifier in question. Fixed and floating point numbers are printed to a standard number of places, while the printing style for integers depends on the current mode, which can be set by a keyboard command. This allows decimal, octal, hexadecimal or ISO character representations of integers.

Stack tracing

Since CORAL is a block-structured language, the effect of restarting the program in an outer block or procedure after a fault in an inner procedure could be to leave rubbish on the stack. This in turn could lead to further spurious errors. To avoid this, a facility is provided to trace back through the sequence of called procedures from the point at which a fault occurred, removing the procedure activation records in the process. This enables the machine stacks to be reset to a state suitable for restarting, and provides the programmer with some useful diagnostic information. Similarly, if a fault is detected in a library procedure, DDS automatically prints a trace of calls until the user program is reached.

CORAL subset for patching

Clearly, the size of the debugging program is of great importance—it was intended to occupy no more than 3K of 16-bit words—and inevitably the subset of CORAL that can be used for patching must be greatly restricted. The aim has been, as far as possible, to remove all the high-level features such as 'FOR'-loops and compound conditions, leaving only the primitive operations that are necessary in any programming language. Only integer arithmetic is allowed. Apart from this, procedure calls and arithmetic expressions retain their generality.

Each patch is regarded as an implicit block, and may be headed by declarations. Labels may appear within a patch, and both local and non-local jumps are permitted. Only one CORAL source statement is allowed on a line, the new line being treated as equivalent to a semicolon, unless a continuation marker 'CT' is used. There is no restriction on the number of such continuation lines. The size of a patch is limited by the space allowed for the patch code and the patching compiler's tables. Each patch is independent of the others, and there is no way of getting at the inside of the implicit block, since patch statements are not numbered. It is impossible, for example, to set a breakpoint within a patch, or to patch a patch. This restriction is imposed to minimise the amount of data that needs to be recorded for each patch.

Table 1 List of commands

PATCHING COMMANDS	BREAKPOINT COMMANDS	STORE EXAMINATION COMMANDS	MISCELLANEOUS COMMANDS
X Alter one or more source statements	B Set a breakpoint at given statement	E Examine a location (symbolic/absolute)	S Set current segment
Y Insert a patch after given statement	C Continue from breakpoint	I Insert a value into location just examined	U Use specified program map file
R Remove patch previously inserted	P List all breakpoint positions	F Examine next location (used after E)	Z Reset debugging program to original state (lose all patches)
	T Set number of times through breakpoint before stopping	A Examine machine register	OUT Remove one or more procedure activation records from the stack
		D Print values between specified locations	
		M Change printing mode	

Even if a patch is deleted, the code generated remains in the patch area, which will eventually become full. At this point the source program must be edited and re-compiled. The size of the patch area is set at load time of the program, and is not fixed.

The syntax of the CORAL subset for patching is given below. Syntax errors detected by the patching compiler will cause the current line to be forgotten. Likewise, the character ! will cause the whole patch to be ignored, so that the user can start afresh.

```
⟨subset statement⟩ ::= ⟨label⟩:⟨subset statement⟩|
                       ⟨conditional⟩|⟨simple statement⟩|
                       ⟨declaration⟩
⟨assignation⟩ ::= ⟨destination⟩ := ⟨expression⟩|
                  ⟨destination⟩ := ⟨procedure call⟩
⟨destination⟩ ::= ⟨wordreference⟩|⟨partwordreference⟩
⟨wordreference⟩ ::= ⟨identifier⟩|⟨identifier⟩ [⟨subscript⟩]
                    |[⟨subscript⟩]
⟨partwordreference⟩ ::= 'BITS' [⟨totalbits⟩, ⟨bitposn⟩]
                        ⟨wordreference⟩
⟨subscript⟩ ::= ⟨identifier⟩|⟨signed integer⟩
⟨conditional⟩ ::= 'IF'⟨expression⟩⟨relop⟩
                  ⟨expression⟩'THEN'⟨simple statement⟩
⟨simple statement⟩ ::= ⟨assignation⟩|⟨call⟩|⟨jump⟩
⟨call⟩ ::= ⟨procedure call⟩
⟨procedure call⟩ ::= ⟨identifier⟩|⟨identifier⟩
                     (⟨actual parameters⟩)
⟨actual parameters⟩ ::= ⟨actual parameter⟩|
                        ⟨actual parameters⟩,
                        ⟨actual parameter⟩
⟨actual parameter⟩ ::= ⟨identifier⟩|⟨identifier⟩
                       [⟨subscript⟩]|⟨expression⟩
⟨jump⟩ ::= 'GOTO' ⟨place⟩
⟨place⟩ ::= ⟨identifier⟩|⟨identifier⟩[⟨subscript⟩]
⟨label⟩ ::= ⟨identifier⟩
⟨identifier⟩ ::= ⟨first⟩⟨rest⟩
⟨first⟩ ::= ⟨letter⟩|$
⟨rest⟩ ::= ⟨letter or digit⟩|⟨rest⟩⟨letter or digit⟩
⟨letter or digit⟩ ::= ⟨letter⟩|⟨digit⟩
⟨letter⟩ ::= A|B|C|D|E| . . . X|Y|Z
⟨digit⟩ ::= 1|2|3|4|5|6|7|8|9|0
⟨relop⟩ ::= 'NE'|'GE'|'LE'|'GT'|'LT'|=|>|<
⟨declaration⟩ ::= 'INT'⟨identifier list⟩|
                  'INT' 'ARRAY'⟨identifier⟩
                  [⟨signed integer⟩:⟨signed integer⟩]
⟨identifier list⟩ ::= ⟨identifier⟩|
                      ⟨identifier list⟩, ⟨identifier⟩
⟨expression⟩ ::= ⟨optional sign⟩⟨unsigned expr⟩
⟨unsigned expr⟩ ::= ⟨primary⟩|
                    ⟨unsigned expr⟩⟨operator⟩⟨primary⟩
⟨primary⟩ ::= ⟨wordreference⟩|⟨partword⟩|
              'LOC'(⟨wordreference⟩)|(⟨expression⟩)|
              ⟨number⟩
⟨operator⟩ ::= ⟨shiftop⟩|
               'MASK'|
               'UNION'|
               'DIFFER'|
               *|/
               +|−
⟨shiftop⟩ ::= 'SRC'|'SRA'|'SRL'|'SLA'|'SLC'
⟨partword⟩ ::= ⟨partwordreference⟩
⟨totalbits⟩ ::= ⟨integer⟩
⟨bitposn⟩ ::= ⟨integer⟩
⟨number⟩ ::= ⟨integer⟩|@⟨hexadecimal digits⟩|
             #⟨octal digits⟩
```

In the syntax above, the arithmetic operators are listed in descending order of priority. The construction 'LOC'(⟨wordreference⟩) yields the machine address of the variable or array element ⟨wordreference⟩, while the converse construction [⟨subscript⟩], called an anonymous reference, obtains the contents of the address which is the value of the ⟨subscript⟩. A partword reference, e.g. 'BITS' [3, 4] A, would obtain the three bit field starting at bit 4 of A, and treat it as an unsigned integer (in the Argus 700, bit 0 is the most-significant bit). The remaining constructions in the subset should be self-explanatory even to those unfamiliar with CORAL, and will not be described further.

3. Implementation

Error handling system

Two versions of DDS exist. These differ in the way in which they communicate with the user program. This in turn leads to differences in the method of handling errors detected in the user program. Entry to DDS can be caused in three ways:

1. Errors detected by hardware.
2. Errors detected by software, e.g. the file management system. Included in this category are interrupts given by the user from his keyboard.
3. Breakpoints.

Breakpoints are handled by planting a special instruction in the user program, which causes a function code violation when it is executed. To the executive, then, all the above cases are treated in an identical manner.

The first version of DDS is loaded as a segment of the user program. Errors are reported to the debugging segment by means of the error vector mechanism. A message containing information about the error (error number and values of specified registers) is placed in the debugging segment's private stack, and the program is re-entered at a preset address within the DDS segment. The re-entry address and initial register values are specified by the debugging segment by means of a call to executive on the first entry to the program. Clearly, with this system, it is not possible to apply memory protection to the user program since the DDS segment is part of that program and DDS must be able to alter the program area to plant breakpoints and patches. Thus the program may in certain circumstances be able to corrupt both itself and DDS.

The second version is completely secure. Here the debugging program runs as a separate task. The Argus 700 Executive (Benson, 1973) allows one task to act as a supervisor (or Program Monitor) to another, and all faults in the user task are reported to the supervisor. Furthermore, all calls to executive are passed to the supervisor for vetting. It is therefore possible to give full hardware and software protection by making the debug supervise the user program.

The main advantage of the first version is that it may be used with foreground tasks (e.g. communication line drivers) that do not have a supervisor because of the time penalties involved.

Structure of the DDS program

The debugging program was designed to be as short as possible. It consists of a short steering program and a number of overlays which actually carry out most of the work involved in decoding user commands, reporting errors, and compiling patches. The structure is illustrated in **Fig. 2.**

For every command the user types, at least two overlays must be brought down from disc and four more for each CORAL statement typed. In addition, several disc accesses may be necessary to interrogate the program map, which also resides on disc. Thus there is a fairly high overhead on disc transfers, but since this disc activity only takes place when the user actually types a patch, the overhead is acceptable. This was the reason for choosing to compile patches rather than to interpret them.

Although an interpreter might have been shorter and simpler to implement, it would have consumed a great deal more

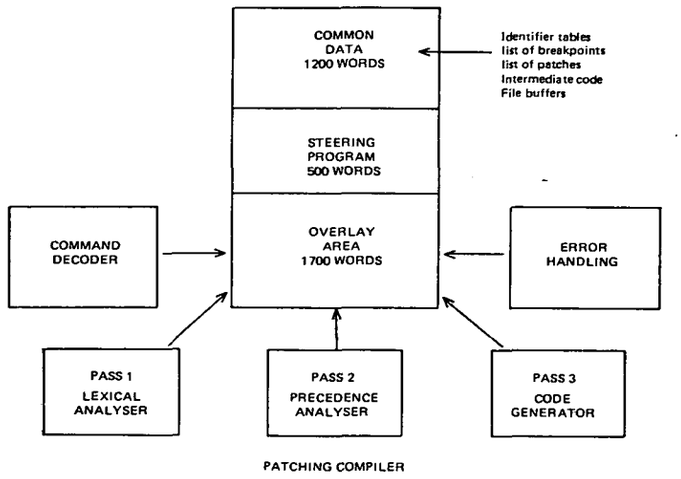

Fig. 2 Structure of debugging program

processor time, and the interpreter would have to be kept with the user program permanently. As it is, the debugging program can be rolled out when the user is not typing commands. An interpretive scheme would also have involved keeping a larger amount of information in core for each patch.

The patching compiler takes three passes to compile each statement. The first pass is a lexical analyser, which reads in the source statement (including continuation lines) and produces a string of internal symbols. This pass also handles labels, and looks up in the program map any identifiers (except labels) which have not been declared in the patch. The first time such a non-local identifier is encountered, its associated attribute information is read into the local identifier table. Thus each identifier need only be looked up once. This scheme avoids needless disc transfers, since the same identifier often occurs more than once in a patch. Any labels used and not defined in the patch are looked up non-locally at the end.

The second pass performs an operator precedence analysis on the line of internal symbols produced by the first pass, and also carries out type checking of identifiers. The third pass is concerned with code generation from the precedence tree produced by the second pass. About 512 words are allotted for the patching compiler's identifier tables, attribute tables and intermediate code areas, allowing patches of upwards of 20 lines to be compiled.

The storage allocation mechanism adopted for the patching compiler is extremely simple. The user patch area is laid out as in **Fig. 3.** There are eight base registers in the Argus 700, of which register 0 is the stack pointer. DDS requires exclusive use of base register 7, and so when compiling a program for DDS, the compiler uses the remaining six registers for the object program. The use of a single base register for patches enables 128 words to be directly addressed by base-displacement addressing. This area is used for identifiers, labels (which must be indirectly addressed), array headers and constants. The first part of the area is used for the 8 breakpoint return addresses, and other housekeeping information takes up a further 16 words. The user is thus able to use about 100 identifiers, constants and labels. The rest of the patch area is used for the compiled code for patches, array elements and housekeeping.

4. Program map

The program map forms the sole interface between the CORAL compiler and DDS, and consists of five tables produced by the compiler at various stages in the compilation. The format of these tables is largely determined by the need for simplicity of production and interrogation. There is insufficient space available in the debugging program for complex searching algorithms, or in the compiler for the creation of elaborate data structures.

The program map consists of:

1. Scope table
2. Identifier table (IDENT)
3. Identifier information table (IDINF)
4. Statement number table
5. Stack position table

One set of such tables exists for each segment in the CORAL program. Keeping segments short will clearly reduce the search time for the tables.

Scope table

The scope table is used to determine which block contains any given source statement. Each block is assigned a serial number, and the table consists of pairs of words as in **Fig. 4.** The table is used in the process of looking up identifiers. It is searched linearly to determine the innermost block in which to look for the required identifier.

Given a statement number:

1. Search forwards until a 'BEGIN' statement number is greater than the given statement number. This ensures that the required block starts before the given statement.
2. Search backwards from the point reached in 1 until an 'END' statement number is greater than the given statement number. The position of the entry reached is the serial number of the required block.

The procedure in 2 is repeated to find the block enclosing that one previously found. This may be done several times if the identifier sought is not in the inner blocks.

Identifier table

The identifier table is produced by the first pass of the compiler All entries with the same block serial number are contiguous. To look up an identifier, the table is scanned until the appropriate section is found, and thereafter each identifier entry is compared with the identifier sought until a match is found or the end of the block section is reached. If the search must be continued in outer blocks, it need only start from after the section previously reached, since innermost blocks appear in the table first and the outer block last.

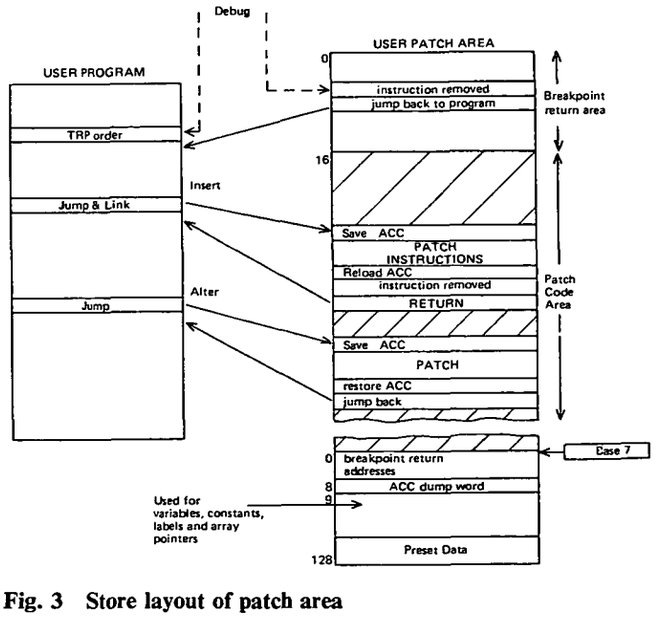

Fig. 3 Store layout of patch area

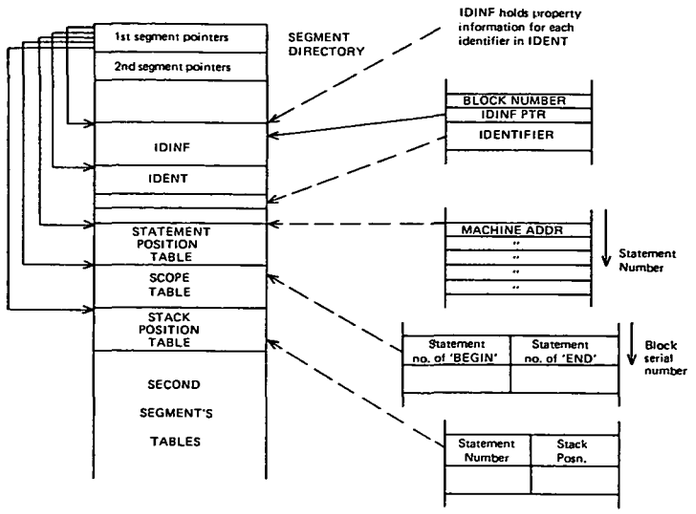

Fig. 4 Program map file format

Identifier information table

The second word of the IDENT table contains a pointer to the relevant entry in this table, which contains property information for all the identifiers in the program. The table is written by the third pass of the compiler, by which time all relevant information about the identifier is known, and addresses are in their final base-displacement form.

Statement number table

This is a simple list of the address of the machine instructions corresponding to each statement number. It is used directly to find the address at which to insert a patch, etc. When a fault is detected in the user program, the address of the fault is handed back to the debug, and the statement number table is searched to find the corresponding source statement so that the user may be informed.

Stack position table

The need for the stack position table arises because the Argus 700 has a hardware stack. Items on the stack such as parameters of procedures are addressed relative to the stack pointer register SPR. Thus the displacements of such parameters may vary from place to place within a procedure (e.g. due to 'FOR'-loop control words which are held on the stack). The table gives the number of words on the stack at the start of each source statement. Statements for which this number is zero (the majority) are not listed in the table.

5. Conclusion

The DDS system was written in CORAL, and was initially implemented on the Ferranti Argus 500. Testing was carried out in an environment which simulated the software system of the Argus 700. These tests demonstrated that the fundamental design of the system was sound, but at the time of writing the final versions had not been tested on the Argus 700 due to the absence of certain other necessary pieces of software. Thus it is not possible to describe any practical experience gained with the system.

The design and development of DDS to the stage described above has occupied one man-year of effort. It is expected that the system will be fully operational by the time this paper appears.

Acknowledgements

The author wishes to thank his friends and colleagues at Ferranti for their help and advice during the project and in the preparation of this paper. Particular thanks are due to P. Bayley, D. J. Pearce, and H. B. Williams.

The referee's helpful comments and advice were also much appreciated.

References

BENSON, D. (1973). *Modular Organiser for an On-line Computer*, IEE Conference Publication Number 102.

CUFF, R. N. (1972). A conversational compiler for full PL/I, *The Computer Journal*, Vol. 15, p. 99.

EYRE, D. M., and WILLIAMS, H. B. (1973). *The Application of CORAL 66 to Control Computers*, IEE Conference Publication Number 102

Book review

Computational Methods for Matrix Eigenproblems, by A. R. Gourlay and G. A. Watson, 1973; 132 pages. (*John Wiley and Sons*, £3·50.)

This book is based on lectures given by the authors to M.Sc. and undergraduate students at the University of Dundee. Its aim is to provide a suitable text for courses on the numerical solution of matrix eigenproblems. The intention is to present the more commonly used and reliable techniques in a concise, straightforward manner, without any detailed error analysis but stressing where necessary the dangers of unsuitable methods. The student thus obtains a good overall view of the subject which, for a fuller appreciation of any topic, would need to be supplemented by further reading in more advanced texts.

The book contains fifteen short chapters. The first three illustrate how eigenvalue problems can arise in practice and give the required background theory and transformations for later use, including material on the solution of linear equations. The next three discuss the power method (including inverse iteration and simultaneous iteration using a number of trial vectors) and these are followed by three further chapters dealing with the methods of Jacobi, Givens and Householder for Hermitian matrices, and the Sturm sequence and inverse iteration procedures for calculating the eigensystem of a real symmetric tridiagonal matrix. Application of the QR algorithm to the latter problem is discussed in Chapter 10 (the LR algorithm gets only very brief mention here) whilst Chapter 11 touches on extensions of Jacobi's method to general matrices. Reduction to upper Hessenberg form and use of subsequent techniques, including in particular the QR algorithm, are considered in Chapters 12 and 13. Generalised eigenvalue problems are mentioned in Chapter 14 and the last chapter discusses (very briefly) available implementations of the methods described in the book.

The ground covered by the book is conventional and the treatment of necessity rather terse due to the book's shortness. A number of exercises are given for the student at the end of each section and the methods discussed are frequently illustrated by simple examples involving matrices of order three or four. (In one of these examples (on page 125) spurious elements appear somehow to have crept into the last row of the matrix during the final stage of a reduction to upper triangular form which preserves information on the determinants of the principal minors.) Little prior knowledge is assumed of the reader apart from a basic familiarity with the fundamental concepts of matrix algebra, such as multiplication, inversion and determinants The book is therefore suited to a wide audience and provides a short and uncomplicated introduction to the numerical techniques which have proved most successful for the matrix eigenproblem. In so doing the authors' goals have been well achieved.

E. L. ALBASINY (Teddington)