The Development of an Incremental Debugging System

A thesis presented in partial fulfilment
of the requirements for the degree of
Master of Science
in Computer Science at
Massey University

Malcolm John M^{c}Donald

1978

Abstract

Debugging is a major area of software development that has received little attention. This thesis starts by looking at work done in the area of bug prevention, bug detection, bug location and bug correction.

A debugging system, BIAS, is proposed to help in detecting, locating and correcting bugs. Three major design goals are established. Firstly, the system should be simple and easy to understand as this will encourage use. Secondly, the system should be general so that it will be available to a large number of users. Finally, it should be incremental as this will save users' time. An incremental language, STILL, is designed to show how BIAS applies to structured languages.

The construction of the system is shown. Each data structure, and how it is used, is described. BIAS uses an interpretive system and runs threaded code on a pseudo-machine. How the threads are interpreted and how they are set up is shown next.

The use of BIAS is shown by following through an example session with the system. This consists of entering a program, editing it, and running it. As bugs show themselves, various debugging commands are used to locate the bugs. The program is then edited, and the corrections linked into the code so that it will run correctly. This cycle is repeated until no bugs remain, without at any time recompiling the whole program.

It turns out that the best way of achieving the design goals is to extend an incremental compiler host to include debugging commands. This gives a clear emphasis to the power of incremental compilers.

Acknowledgements

I would like to thank everyone at the Massey University Computer Centre and all those other people who have listened to my ideas, so often with enthusiasm. Just explaining the system has greatly aided me in the development of the system described herein. In particular, I would like to thank my supervisor, Ted Drawneek, for always having a co-operative ear, and for all the guidance and constructive criticism that was of such great value. Thanks also to Margaret Dench for her excellent job of typing when time was short, and to Phill Jenkins for general discussions about programming, which are always useful, and for the artwork on the figures.

Table of Contents

List of Figures

Chapter 1

DEBUGGING - AN INTRODUCTION

"Bloody instructions which, being learned,
Return to plague the inventor."

Macbeth in Macbeth I (vii)

When Macbeth was plotting the murder of Duncan, he realised that his plans would eventually turn against him. He decided not to go ahead, but, being easily led, he did the deed and his original thoughts were proved correct. What he needed was a good debugging system so that he could correct his mistakes before they became fatal.

Debugging has been around since Whirlwind I [Schw 71, VanT 74], yet it is one of the most neglected areas in software development [Bern 68, Gris 70, Pier 74]. This is certainly due in part to 'debugging' being a dirty word. No-one likes to admit that they make mistakes, so when the time comes to correct them, people tend to hide the fact. Consequently, each programmer thinks that he is the only one who takes such a long time to do the job, and that there is little general need for debugging aids.

What is debugging? Testing and debugging are often confused with each other as they usually overlap. When a program compiles correctly, the programmer enters data in a testing phase in which errors are detected. The programmer then tries to locate and correct these errors in a debugging phase, and the cycle is repeated. As time passes, the testing phases get longer until there appears to be no bugs (although this is often not the case). Of course, the debugging phases generally do not get shorter, and may well get longer as the errors become more obscure.

So here we arrive at a major point. Debugging takes more time than any other aspect of programming. Estimates vary from 30% to 90%

[Gain 69, Goul 75, VanT 74], so it is clear that to improve software production time, debugging is a good, if not the best, area to attack. This thesis will show the development of BIAS (Batch and InterActive System), a debugging system which collates, clarifies and simplifies existing systems.

1.1 Bug Prevention

Prevention is better than cure is a proverb well suited to debugging. While it is unlikely that all bugs can be prevented, any technique that can reduce their number or their complexity is welcome.

Every program should be well designed. This is best achieved using a top-down technique such as step-wise refinement [Wirt 71]. The modules produced should be of limited size [Your 75] and be able to stand on their own as far as possible. Interfacing is thus kept to a minimum which not only reduces the chance of having bugs, but also reduces the scope of any that do appear. This is known as bulkheading [VanT 74]. Debugging is made easier as bugs are isolated and much less likely to interact.

Style is a mark of individuality that pretends to excuse many faults, but like any writer, the programmer must use style as a beacon not a smokescreen. Good style not only reduces the number of bugs but also makes debugging much easier. One major technique is the selection of identifiers. This is the most important principle in program readability [VanT 74], although comments saying why something is done rather than what it is doing are still essential. Structuring and indentation are also valuable aids [Dora 72]. Ultimately, whatever features of style are used, they must be used consistently.

Compile-time errors are much easier to prevent than run-time errors. The prevention of compile-time errors can be done with interactive systems. There are two methods currently in use. Incremental compilers get the user to correct his syntax line by line [Ryan 66]. Interactive text editors such as EMILY [Hans 71] and GENISYS [Barr 75] actually prevent the user from making errors. EMILY works from any BNF grammar and so prevents only syntax errors, GENISYS and the system described by Lasker [Lask 74] perform static semantic checks as well.

These text editors all work by building a parse tree from the BNF grammar. All possible productions for each non-terminal are displayed and the programmers selects which production he wants by sending its associated number [Barr 75] or by pointing a light-pen to it [Hans 71, Lask 74]. The syntax of a program must consequently be correct. Unfortunately programs require a long time to enter by this method. Entry of identifiers (all the systems mentioned) and expressions (GENISYS) by typing them directly in does help. This heads the idea back towards incremental compilation.

1.2 Bug Detection

Bug detection is finding out if there are bugs. The usual tool for this is testing. As the number of possible data sets is usually astronomical, exhaustive testing is impractical, but one can improve reliability and shorten production time by using carefully selected test-cases [Buxt 69, VanT 74]. Testing should first show that the general case works. Extreme data, exploring the fringes of what is acceptable, are then tried to make sure no overflows occur. Finally, exception conditions for data that is blatantly or marginally wrong are tested to make sure errors are reported. Each type of data (general, extreme and exception) will cause its own type of error which should help to pin-point bugs. Whatever type of data is used, it must be easy to predict what output will result. If not, it will be hard to locate the bug, which may even be in the prediction of the output.

Modular test-beds allow modules to be tested individually. All globals are set by some device, and all calls from the module will be dummies. With an interactive system, the programmer can perform the action of dummy subroutines himself. He can also change parameters while the program is running and probe the boundary conditions more effectively. Bate describes such a system which resembles a breakpoint debugging system [Bate 74]. With this system, variables can be examined and altered, and breakpoints set to give the tester control anywhere in the program. A design goal for BIAS was to facilitate module testing (see section 2.5).

Two novel methods of finding bugs in systems software have been

developed by Rain [Rain 73]. The 'Bug Farm' randomly alters correct data, often providing unthought of combinations. The 'Bug Contest' offers an incentive to users to find bugs. This leaves the programmer free to repair bugs without having to spend hime time on testing. It also has a useful side-effect in that user reluctance to try the new software is overcome, and the new system is used to its fullest extent.

Unfortunately, testing shows the presence, not the absence of bugs. A formal proof of correctness, however, can show that a program is error-free, and if it is not, it can help pin-point the error(s). A bug in the F-level PL/I compiler was found by such a proof where testing had failed to detect it [Buxt 69]. Formals proofs can eliminate testing and simplify debugging while ensuring correctness. Because of this, correctness is an important area of study and is receiving much attention [Elsp 72, Lond 70].

Lowny commented that "ANY significant advance in the programming art is sure to involve very extensive automated analyses of program." [Buxt 69]. The verifying compiler described by King is an example of this [King 71]. Predicates are submitted to the compiler with the program, and the compiler does the proof. If the program is not correct, the likely source of error is pointed out. The main drawback is that as with all predicate proofs, the predicates are difficult to formulate. If the program is written with the proof in mind, the predicates will be easier to produce [Dijk 69], but even then the proof may well explode with program size.

There are compromises between formal proofs and the classical methods of debugging and testing. Less than rigorous formal proofs, amounting to a kind of disciplined desk-checking will often yield many bugs [Schw 71]. Stepwise refinement is a very informal method of proof that tends towards prevention rather than detection [Wirt 71]. Proving that critical parts of a program are correct will prevent many bugs without excessive overhead. The same will be true for proving that certain anomalies are not present in a program. Such a system is DAVE [Oste 76]. DAVE detects two types of anomalies: reference before assignment and assignment followed by no reference. This picks up uninitialised variable errors and also helps to find spelling mistakes.

All the methods of proof mentioned so far do not execute the program. However, predicates can be used at run-time to check for data anomalies. 'ON' statements in PL/I and Burroughs Extended Algol cause only low-level faults to be trapped, but are still very useful in detecting errors. Algol W 'asserts' are at a much higher level and can be as sophisticated as formal predicates [Satt 72].

1.3 Bug Location

Once a bug is known to exist, the next step is to find out exactly where and what it is. The commonest method of locating bugs is to pore over a program listing, doing a mental desk-check with the data that made the program go wrong. This is particularly true of small routines which the debugger did not write [Goul 75]. However, with larger programs, some bugs can be very hard to find without more sophisticated tools.

One of the oldest debugging tools is the core dump, usually taken after the program had made an error that upset the hardware. The entire contents of memory and all the registers would be printed in hex or octal, with very little to signify what was what. With the advent of high-level languages, the meaning of such dumps became more obscure, although they are still used [Blai 71, Gris 70, Kuls 71]. However the trend is to format dumps so that they relate to the source program. Stack dumps on the Burroughs B6700 are easy to follow when used in conjunction with the compiler option STACK (which is available for all major compilers). The B6700 also has a dump analyser for core-dumps which makes them much easier to understand. Selective and snap dumps are more useful to the high-level language programmer. Dumping suspect variables at carefully chosen places in the program will give a lot of information with relatively little output. [Ferg 63, Gain 69, Satt 72].

Tracing is another old debugging tool. There are many kinds of trace, but they all show where some trace condition or trap has occurred. Typical traps are storing to a variable (store trace), change of flow of control (flow trace) or reaching a given line of code (line or source trace). Traces are very useful as they can give a full history of program execution. They can, however, generate a lot of superfluous

output if they are not controlled [Grov 74]. Dynamically turning the trace on and off [Gris 70, Gris 73], tracing only when a dynamic condition is true [Blai 71, Ferg 63] or limiting the trace by a static condition or loop [Burr 74 A, Burr 74 F, Satt 72] will reduce output considerably. With interactive systems, the trace can even be turned on or off by the programmer [Bull 72, Kuls 71].

Program statistics are useful for deubugging and increasing efficiency. The simplest statistic is the execution time for the program. This in itself can often help to locate errors. The execution summary of SNOBOL 4 [Gris 73] is an extension of this. Burroughs Algol has a compiler option to print the time spent on each procedure of a program. MUSSEL and Algol W go one step further and give the execution count for each statement [Grov 74, Satt 72]. A different kind of statistic is the cross-reference listing, [Brow 73, Burr 74 A], which is particularly useful when looking through large programs.

All the tools mentioned so far are batch-oriented. With interactive systems a much greater range of tools can be made available. The basis of most interactive debugging systems is breakpointing. A breakpoint is a device for giving control to the programmer at an interactive terminal. The programmer can then converse with his program, see exactly what it is doing, and even correct it if it is wrong (section 1.4).

Interactive debugging in the days of machine-code programming consisted of stepping through the program instruction by instruction until something looked wrong. By using console switches a correction would be made and the process of detecting and locating bugs would continue. Stepping statement by statement is the high-level language equivalent, and it is just as useful [Gain 69, Pier 74, Burr 76 A]. This method can be extended by stepping several statements at a time [Pier 74] or running the program at an observable speed [Bate 69].

Bugs are usually detected some time after they occur, which makes them hard to locate. This is especially true of evanescent bugs which by Murphy's Law never seem to occur when they are being looked for. If, however, the program can be backed up to the point of error, the

problem is solved. Reversible execution can be implemented in two ways. The simplest is by checkpoints [Bate 74, Gris 71]. This requires little effort, but a forward execution from the checkpoint may differ from the original and evanescent bugs might not appear again. The other method is to record each change in the program as it happens [Bate 69, Davi 75, Zelk 73]. This takes a lot of process time, but using the history file for forward execution when possible will guarantee the same execution path will be used.

It is often useful to run a program again without changing its variables. This is an important feature of an incremental compiler called incremental execution [Bull 72, Rish 70]. Incremental compilers can also run the program from any point in the program, or even run an incomplete program. This greatly facilitates debugging as small sections of code can be tested independently of the rest of the program.

There is a right way and a wrong way to use any tool, and debugging tools are no exception. The programmer should get an idea of what is wrong with his program and carefully select his tools rather than apply battering-ram tactics. It will be cheaper, quicker and cause less headaches. Knowing what errors might occur is a start [Brow 73, VanT 74]; certain errors are best located with certain tools. For example, bad initialisation can be easily found by a store and/or fetch trace while looping errors can often be found by using execution counts.

1.4 Bug Correction

Once a bug has been located, it must be put right if the program is to work correctly. Bug correction usually consists of patching the source program and recompiling it, although only the offending subroutine may be recompiled and then bound or link-edited to the rest. In either case, the program must be run again from scratch. This approach is time consuming to the programmer and to the machine. The alternative is to correct the error at run-time.

When an exception condition occurs, the computer will detect it and terminate the program. However, if PL/I or Burroughs Algol 'ON' statements are used, the program can retain control. This will enable the program to make some correction or output suitable for debugging information. At any

rate, the program can continue execution, which may lead to finding other bugs.

Such interrupts are used for hardware and related errors, but software errors are not so easily detected or corrected. Recovery blocks are a solution whereby a section of code will ensure a condition, which is like a predicate in formal proofs [Rand 75]. If the main piece of code, the primary alternative, does not satisfy the condition, other alternatives are executed until the condition is satisfied or there are no more alternatives, which causes a fatal error. This allows the program to continue and give reasonable results even it if is not completely correct.

Conversing interactively with a running program, inspecting and changing its variables is a powerful tool [Barr 69]. If a programmer can watch his variables change value, he can actually see his program go wrong and possibly see why it went wrong [Balz 69]. In any case, correcting wrong values will temporarily patch the program. The ability to do this is the basis of most breakpoint debugging systems [Appendix A] and is an important advantage of time-sharing.

If the programmer has to correct the same error each time it is executed, debugging will take a long time. By making a run-time patch, he actually changes the program so that hopefully the error is fixed for that execution of the program. Unfortunately, run-time patches on most debugging systems often do not resemble the source language of the program and so are not permanent [Bate 74, Blai 71, Gris 70]. Also, they are usually very limited as to what they can do (assign only constants [Bate 74, Burr 74A], no conditional statements [Ashb 73]).

Incremental compilers solve all these problems. Patches have the same status as any other part of the program. They are in the same source language and suffer no restrictions as the same compiler is used for patches as for the rest of the program [Bull 72, Ryan 66]. When an error is located at run-time, a patch can be made, linked into the rest of the program and the same run continued as if nothing had happened. Editing the program at run-time is an important aspect of incremental compilers [Rish 70].

So far, the programmer suggests the correction, tries it and lets the computer find out if it works. Davis suggests that with inexperienced programmers, it is better for the computer to suggest the mistake as well [Davi 75]. When an error occurs, the computer backs up the program showing how the error was reached and what could have caused it. When the user decides on the cause, the computer explains how it could fix the mistake, checks with the user and does the fix.

Chapter 2

DESIGNING THE SYSTEM

"The best laid schemes of mice and men
Gang aft a-gley"

To a Mouse Burns....

2.1 Some Possible Approaches

A debugging system is a collection of debugging tools and a method of binding them to a program. The debugging tools vary greatly but they always interface to the program via some kind of trap (Sect. 1.3). When a trap is executed, control is passed to the debugging system which evokes the required tools.

Traps can be inserted by four basic methods. A precompiler can add source language traps to the source program before compilation [Balz 69, Ferg 63]. A source statement is inserted at every place a trap may be required. The statement transfers control to the debugging system, which must determine why the trap was caused. Each pre-compilation needs computer time, and compilations are larger because of the extra source. However, fairly efficient object code can be produced. A major advantage of the precompiler approach is that no alteration of existing software is required, although obtaining symbol tables and such from a compiler makes the system easier to write. A major disadvantage is the necessity of reserving some identifiers for the debugging system which will make some programs unacceptable to the precompiler.

A compiler can insert traps in a similar fashion to a precompiler, the main difference being that there is no intermediate source, so no reserved identifies are required. There is little extra overhead in computer time as compared with normal compilation. Efficiency is about the same as for a precompiler system as similar code is produced. The main drawback is that modifications would have to be made to an existing compiler if it was used [Burr 76, Pier 74, Satt 72].

Traps can be inserted into normally compiled object code with physical breakpoints [Gain 69, Wolm 72]. The breakpoint physically replaces a portion of code which consequently must be executed from the debugging system. The code produced is very efficient unless a large number of traps are inserted. Although code produced by an ordinary compiler is used it is usually necessary to tamper with the compiler to obtain the relationship of the source to the object.

Traps can be put into the machine rather than into the code. This is particularly true for the software pseudo-machine of an interpretive system [Bate 74, Grov 74]. Special debugging instructions can be added to the pseudo-machine. Also any specific instruction can be trapped by modifying the pseudo-machine. Systems that rely solely on a software machine are slow, so some interpret machine-code, modified slightly to aid debugging. They can thus run any part of a program that has been debugged on the hardware machine [Blai 71, Gris 70, Kuls 71]. Of course, there is no reason why instructions cannot be trapped by firmware or hardware.

The method used to insert traps involves a trade-off between efficiency, ease of implementation and the power of the debugging commands. Many debugging systems were built with all of these as design goals [Asb 73, Balz 69, Gris 70, Satt 72]. This has to lead to compromises. With batch systems, the major cost of a run is the processor time, so efficiency is more important than power. With interactive systems, log-on costs will probably dominate, so having a powerful command language will be more important as it will reduce log-on time.

Many debugging systems are written for interactive use (Appendix A). Since it has been claimed that users get sloppy [Buxt 69] and spend more time on a program [Carp 76] when working interactively, this could be a bad thing. With poor system response and a heavy demand for terminals, users do waste a lot of time. They will wait around for considerable periods for the system to respond or to get onto a terminal, and if the system is slow they will try to speed up and try things without thinking and thus make mistakes. In this situation both arguments against interaction are valid.

It must be realised however that if response is good, and the user can leave the terminal for a short time and come back to find it available then interaction comes out in a much better light. A survey of batch/

interactive comparisons by Sackman [Sack 68] has shown that interactive users can develop a program in less working time than batch users, although the cost is the same either way as batch users require less computer time. It also appears that programs subjectively prefer interaction, although more work needs to be done in this area.

2.2 Design Goals

Up to now, the design goals of most debugging systems have been aimed at getting an efficient system available quickly. They will try to make a system that is easy to use, but this is often incompatible with the other goals, and so is compromised. The result is that although the user can refer to his program at the source level, many of the command constructs are complicated and confusing. Unfortunately, this causes user resistance to the system as was found by Grishman [Gris 71] . The author has also experienced this difficulty with Burroughs Algol break-point debugging. It tends to become a last resort because of restrictions and clumsiness.

To prevent this happening, when designing a system the first consideration should be the user. The commands should be powerful, easy to understand, easy to use, and easy to learn [Evan 66, Pool 73]. Then and only then can implementation restrictions be considered. Since the command language will already be decided, and restrictions will not affect the simplicity of the commands, only their power.

When designing BIAS, the implementation was only a secondary goal. Producing just another debugging system, though of immediate use, would merely solve the technique for one language on one machine. It was hoped that some software would be running on a limited scale, but the main aim was to design a system that was neither language or machine specific.

The basic goals were to make the system:

(1) Simple - the command language should be simple and contain as little syntax as possible, while giving as much power as possible. It should be easy to learn and it should prevent small, typographical errors from causing the loss of control of the program, (Section 2.3).

(2) General - the basic design should be language independent, machine independent and program independent. The algorithms should readily be converted to different programming languages. The system should be able to accept any program written in a host source language, and should run in both batch and interactive modes. (Section 2.4). The commands should apply in a consistent manner to every program construct, and not have any special cases or unusual restrictions.

(3) Incremental - patches should be made at runtime in the host language and the program should be automatically updated before the system relinquishes control. The program should also run incrementally under the debugging system. The means that the system can run as an incremental compiler and as a modular test-bed. (Sect. 2.5).

2.3 Simplicity

The command language of BIAS can be divided into three parts: those that affect the program, those that affect the data of the user's program and those that help the user control his program. These are called editing commands, immediate commands and trap commands respectively. (See Appendix B for syntax). They can all be used at any point that the command analyser has control as long as this is consistent with the program. This precludes absurdities like continuing the program when it is not even running.

The editing commands are a bare bones and could easily be extended. The commands allow a user to retreive a program, or make a new one. He can insert new lines, delete old ones and list any part of the program. He can start the program running, or if it is stopped at a trap, continue execution at another part of the program. Finally he can either discard the program or save it, in which case the code and other data concerning the program will be saved in the same file as the source. Since this thesis is primarily concerned with debugging, this basic minimum is all that was provided, but on a larger system this would undoubtedly be extended to more powerful commands.

The "immediate commands" have the same format as statements of the host language. The statement is compiled as if it occurs at the place the program is stopped at. Most useful will be the write statement and the assignment statement which will enable the user to examine his program's data and modify it. The procedure call or invocation statement will also be useful for testing individual subroutines without setting up the whole environment for it.

The host source format must make the command language easier to understand as special commands are not required. Also the full power of the host language can be used, including the built-in protection features; e.g. a programmer is syntactically prevented from storing a data value into a procedure identifier etc. [c.f. Gris 70, Kuls 71].

The most important commands are the trap commands. They enable the user to insert, suspend and delete traps. Traps are inserted by using either the BREAK or TRACE command. The only difference between the two is that if the trap is inserted by the BREAK command, the system returns control to the command analyser, whereas TRACE will continue execution. Both report where the trap occurred and the execution count for the statement if required. BREAK could have been done using a BREAK statement in the language, but extending the language was considered undesirable.

Traps are set at all the places indicated by the trap-part. The line-num-seq-LIST gives a range of line-numbers so that only parts of the program may be affected. The default is the whole program.

The statement-type-LIST defines which statements within that range should be trapped. Thus a general store-trace can occur by the command TRACE ASSIGN. Certain statements such as declarations cannot be trapped, as this has no real meaning and there might be no associated code. This will, of course, vary from language to language, so different options will be available to cope with this. ASSIGN, CALL, and IF should be nearly universal, whereas CASE and LOOP (any iterative statement) might not be quite as common. Because it might well be impossible to trap a statement after it has been executed (e.g. GOTO,

RETURN), a trap is inserted before its associated statement. At times traps will be more useful after the statements, but since consistency is essential for ease of understanding, this choice was made.

The other form of trap-part uses a variable-list. This results in the named variables being store-trapped. Any variable can be trapped; in particular, array elements as well as whole arrays can be trapped. Whether the traps will last only while the scope lasts or for each time that scope is used is a semantic problem left to the implementation.

Traps can be controlled by the control-part, which can consist of up to three parts. Firstly, the ON condition will prevent the trap from being reported unless the condition is true. The condition has to be semantically correct at every place it is used. The restrictions thus imposed will vary from language to language. BASIC will have none. With FORTRAN a trap with an ON control will have to be restricted to one subroutine as the symbol table is invalid elsewhere. With a structured language, the trap must lie in the scope of the identifier with the highest lexical level that is used in the condition. The BY-part of the control allows stepping through the program by a specifed number of traps. The FOR-part restricts the number of times the trap will stop. The expressions for each part of the control-part are any valid expression in the host language that derives the appropriate type of value (boolean for ON,integer, or, something coercible to integer, for BY and FOR).

example 1. Algol

```
100    BEGIN
110    REAL A;
120    A := 1;
130        BEGIN
140        INTEGER I;
150        FOR I := 1 STEP 1 UNTIL 10 DO
160            A := A + I;
170        END;
180    PRINT (A);
190    END;
```

Assume the program has reached a breakpoint at line 150. The command "BREAK ON I = 10" is valid only in the range 130-170. Since the whole program is the implied range, a warning will be given. However, had the program just been trapped at 120, there would be an error as I is not even declared at that point. The command "BREAK ON A MOD 2 = 0 BY 2 FOR 2" should not be valid for line 100, but is allowed as otherwise the whole program could never be a valid range. The effect of this command is as follows:

I	1	2	3	4	5	6	7	8	9	10
A	2	4	7	11	16	22	29	37	48	58
trapped?	No	Yes	No	No	No	Yes	No	No	No	No

The program is thus trapped a total of two times (FOR 2) every second time (BY 2) that A is even (ON A MOD 2 = 0)

Each trap can have a statement attached to it. Selective snap dumps are taken by attaching a write statement to the trap. This allows the user to have dumps made using his own formats if desired. At any rate all I/O is done with the same syntax as the host language so that the user does not have to learn any new syntax. A statement attached to an ON condition allows software recovery of data errors in a manner similar to recovery blocks [Rand 75].

All code generated by the BREAK and TRACE commands has one restriction - no references to user subroutines are allowed because these subroutines may have traps set in them. When the code is executed, a trap may occur preventing proper execution of the command. From the user's point of view this means he cannot use subroutines in defining what action he wants at trap. Users of linear languages such as BASIC and FORTRAN will suffer from this as they have only one statement attached at a trap, whereas a subroutine would allow them more. A structured language user can have several statements by making the statement a compound statement or a block.

Traps can be turned off by using the UNBREAK and UNTRACE commands. These will permanently remove all traps of the specified command type that are set by the trap-part following. Consider the effect of the following commands on Example 1.

BREAK	140 - 180
TRACE	120 - 150
UNBREAK	130 - 170

After the TRACE command, lines 140 and 150 will be trapped by tracing, not breaking, so when the UNBREAK command is executed, traps will be removed from lines 160 and 170.

Traps may be suspended with the RESET command. Suspended traps are still present, but are not acted on. When the appropriate option is SET, traps that have been suspended come into effect again. Suspended traps are affected by other trap commands. If traps are suspended, old traps can still be removed, and any new traps will also be suspended. This allows the user to make a production run without removing traps which he may have taken some time to insert. He can thus restore his traps for another debugging run very easily.

The last SET/RESET option, XCOUNT, allows the user to take execution counts if he wishes. When XCOUNT is set, the execution count of any statement which has been trapped is given when the trap is reported to the user. When XCOUNT is reset, all execution counts are also reset to zero. The user can access individual execution counts with a predeclared array called XCOUNT. This is a single-dimensioned array indexed by the line-number of some line of source, and it may be both read from and written to.

2.4 Generality

Language independence

The main form of generality aimed for was language independence. Most conventional languages can run under the system with little or no modification to either language or BIAS. Certainly the command language was designed with this in mind

In order that the statements of the source program can be uniquely distinguished, every line of source has a line-number. This is fairly

common: columns 73-80 of the card is reserved for such a purpose in most languages. However most languages allow more than one statement per line, and even though this is bad programming practice, programs will be written using this feature. However, BIAS requires at most one statement on a line. Since this merely enforces good programming technique, the price is a small one and the advantages outweigh the disadvantages. However, statements are not restricted to one line as this would make some statements almost impossible to write, especially if the program is well-structured with indentation. Note that this does not interfere with the syntax, as the only alteration required to any program will be to start each statement on a new line.

Machine independence

Writing large, machine independent programs is difficult in the extreme. The program would almost certainly have to be written in FORTRAN which would make it longer, harder to write and harder to understand than if it was written in a structured language. To solve this, all programs are interpreted. 'Compilers' translate the source into a pseudo-machine code which is interpreted on a pseudo-machine. The pseudo-machine is relatively short and very simple, and should probably be written quite fast for any machine. In fact, it is highly suited to being coded in assembler, which would greatly aid efficiency. The compilers can then be translated into a language available on the system and compiled with that language's compiler and actually run on the pseudo-machine. This would be fairly slow but does aid portability.

Program independence

After experience with Burroughs Extended Algol's breakpoint debugging system, it was clear that no restriction should be placed on the program by the system. (In the latest software release, the major restriction of the Burroughs system has been removed). In fact no noticeable flaw of this kind in BIAS have been found yet. The only restrictions have been on the commands where procedure calls are invalid in certain places (section 2.3).

Generality of use - batch and interactive modes

Although incremental systems are usually used only interactively,

it was decided to make the system available in batch as well. This turns out to be the major hazard to portability - all machines handle I/O differently. On the Burroughs B6700 all types of file can be easily interchanged, but this is not true of many machines. Since card and remote files are interchangeable on the B6700, BIAS can run easily in both modes, satisfying a major design goal.

Running under batch produces interesting effects as the program does not have to be recompiled for a run. As with many large compilers, patch statements can be merged, but under BIAS these are all that are compiled. The patches are linked into both code and source, thus saving some machine time. When the run is finished, if the program is saved, the updated version will be saved, not the old version, so the whole program will be up-to-date. Any debugging commands, however, will be acted on during the run, but neither they nor their effect will be saved. Debugging commands therefore, come in debugging packets and so are easy to separate from the program.

2.5 Incrementality

This is a vital feature of BIAS which is of immense help in reducing debugging time and machine time. Any patch to the program is made at runtime and the source and code for the patch is linked into the rest of the program. As just mentioned, this saves machine-time as the program need not be completely recompiled. This also saves the user a lot of time as he can continue using the program with only minor modifications to the data.

This is another reason why interpretive methods were used. It is not easy on most machines to fiddle with code in the required manner. However, by making a machine from scratch, this failing can be overcome. Incremental compilers require a threaded code system of some kind. The threads are embedded in the pseudo-machine itself, which increases efficiency considerably. Also, the machine deals with the traps, not the control program. This is explained in Section 3.1 and 3.2. Apart from these special requirements, the pseudo-machine is a stack machine based on the Burroughs B6700. This has the advantages of making code very easy to generate and requiring very little space for code as against a register to register machine. This is especially true for structured languages where the structure is not static.

Other languages adapt fairly easily to this system, but lose efficiency. Since efficiency is not a goal, the stack machine was deemed appropriate.

An alternative method would have been to use mixed code - machine code and interpretive code. It was hoped that this approach could be used, but although the code is much more efficient, it is extremely difficult to arrange the two sections of code and the control program in a workable configuration. The problem is made much worse with a structured language. With a mixed code set-up, the controller has to be a subroutine called by user's program. This gives it only one return point, thus making it impossible to return directly to another part of the program, which is, of course, essential if lines are to be deleted. The return can be made indirectly via go to statements, in which case a sort of threaded code method results, although a very tricky and untidy one. Accessing the data is not easy either, a special highly machine dependent intrinsic being required. The user's program must be a subroutine of a master program which declares the necessary globals, including the control program. The control program must still contain code to translate host source into pseudo-code for patches, as well as a pseudo-machine to execute these. In spite of the gain in efficiency, the cost in effort is too great for the savings, and for these reasons, a mixed code approach was abandoned.

As a result of the method used, BIAS is not only a debugging system but also an environment for incremental compilers and a modular test-bed. Any module (procedure) can be run with only its globals declared. Any procedures the module calls can be empty, in which case a trap is caused allowing the user to alter the data as the missing procedure would. The effect of the module on the globals can be checked at any point.

2.6 STILL

STILL is a STructured IncrementaL Language based on Algol 68, designed especially for the purpose of demonstrating BIAS in action on a structured language. The syntax can be found in Appendix C. Its most important feature is that a semicolon marks the end of scannable

text on that line - anything after it is considered to be commentary - and is equivalent to there being no more text on that line. It is this feature that makes the language distinctly incremental and suitable as a debugging host. If a continuation is required, it must be specifically requested by an ampersand. Again, text after an ampersand is considered to be commentary.

An end of line or semicolon is required at every place that a trap may be required. As a result, statements which contains statements such as CASE and BEGIN are not written on one line. Rather the controlling or identifying part is written on one line and the statements within it are each written on a new line. For example:

Example 2. STILL

(2a)
```
BEGIN  A := 1; B := 2  END
```

(2b)
```
BEGIN
A := 1;
B := 2;
END;
```

(2c)
```
BEGIN;
A := 1
B := 2;  A COMMENT
END
```

(2a) is invalid as each new statement is not on a new line. (2b) and (2c) are both valid in STILL.

Since the production of STILL was not a major goal, the power of the language is currently only on a par with Algol 60. There are three basic types - int, real and bool. Identifiers can be made read only (constants) or initialised in declarations in a similar way to Algol 68. Arrays can contain up to 15 dimensions and it is the number of dimensions that determines the type for parameter passing (c.f. Pascal). Procedures are not initialised on declaration. A proc declaration declares the number and types of the parameters and the type of the procedure only. A proc variable can have a routine assigned to it during execution. Any formal parameters are declared at the head of the routine and must correspond in number and type to those of the variable it is assigned to. Parameters can be of any type except that a formal procedure may not have a parameter of type proc. Parameters can be called by value or reference as in Algol 68, except arrays which must be called by reference and procedure which must be called by value.

```
Example 3.   STILL
     (3a)    REAL PI = 3.14159,A,B,ALPHA := "A";
     (3b)    [0:10, 7 : 8]  INT ARRAY1,ARRAY2;
     (3c)    PROC (REAL) VOID SUBROUTINE,P;
                        . . .
             P := (REAL A) VOID :        (followed by the body of
                                          the procedure)
```

The language contains no GO TO statement. A block may be exited using the RETURN statement. Several levels may be exited by labelling a block and using RETURN FROM label. A routine is also given a value by the RETURN statement. Loop statements are actually implicit blocks, with the iteration occurring within the block. If a control variable is used, it must be declared outside the loop so its value is available both inside the loop and after the loop is exited and can be changed within the loop. The increment and the bound (if present) are both fixed on entering the loop. A loop may also be exited with a RETURN statement.

STILL I/O is fairly simple. Variables can be input and expressions output, but there is no list structure in I/O lists. To write out arrays, a loop statement must be used. The SAME option in both I/O statements allow the same record to be written to at the next I/O statement. Formats are attached to the list elements so there can be no confusion, matching long formats with lists. Several formats are available and although these are not extravagent, they should be sufficient for most needs. Only a field width is required for any of the formats making layouts easy; real numbers are printed in the most suitable form according to the field width.

Expressions are handled by a simple priority mechanism as in Algol 68, but are known to be evaluated from left to right where this does not conflict with priorities. The priorities of binary operators are:

operator	meaning	priority
!	exponentiation	8
*	multiplication	7
/	real division	7
DIV	integer division	7
MOD	remaindering	7
+	addition	6
-	subtraction	6

operator	meaning	priority
<	is less than	5
<=	is less than or equal to	5
>	is greater than	5
>=	is greater than or equal to	5
=	is equals to	5
<>	is not equal to	5
IS	is the same as	4
ISNT	is not the same as	4
AND	logical end	3
OR	logical or	2

All unary operators have a priority of 10. An operand can be preceeded by any number of unary operators which are performed on the operand from right to left. The operators are:

- (negative), + (positive), NOT (logical not), REAL, INT, BOOL, ENTIER (type conversion), SIN, COS, ARCTAN, (trig functions) and SIGN.

Chapter 3

BUILDING THE SYSTEM

"Del dicho al hecho hay gran trecho"

It's a far cry from speed to deed

Don Quixote - Cervantes

Once the syntax for BIAS and STILL was determined, the implementation had to be considered. The first code written was for the pseudomachine. This was initially a straightforward absolute addressed stack machine. Provision for code threading was not made until the requirements were clearer.

The main data structures were designed in parallel, with frequent references to examples to make sure that they would work unambiguously. With a rough idea of the code and the structures, the STILL compiler and the BIAS interpreter were written and tested. These showed what was needed to thread the code and all the programming that had been done was modified accordingly.

Finally, a method of linking the various parts together - pseudomachine, compiler and interpreter - had to be worked out. Again small changes were made to previously written code, but at this stage program organisation was complete. This chapter will show exactly what organisation was used for each part in the final implementation.

3.1 Data Structures

There are three related sets of data structures. The most important are the statement information tables, as they provide the key to the operation of the debugging commands. The symbol table, although structured, is designed so that all identifiers are stored permanently, even outside their scope. This is essential, as during execution any block may be entered and all identifiers valid there must be available to the user. The symbol table and statement information table are linked by the structure table. There are also structures for controlling traps. These are pointed to by the statement information table but not vice-versa.

3.1.1 Statement Information

Each statement of the source program has an entry in the statement information (STINFO) table. Each entry consists of several fields, some of which are an integral part of the machine. These fields are the ones that control code-threading and traps. Fig. 3.1 shows all the fields for a sample program except the trap information.

WORKFILE:

This is not actually a part of the STINFO tables, but it does contain information about the statement - the actual source of the program. This includes the line number of the statement. The workfile can be accessed randomly or sequentially.

RECNUM:

This is an index into the workfile and is the record number of the first line of source for a statement. In fig. 3.1, it is always one less than the statement number, but this will not be true if some statements are continued or if certain patches are made. When access to the workfile is required, the RECNUM of a statement is used to put the correct record of the workfile into the file buffer. Reading or writing the record can then be done sequentially. This is useful for handling continuations as a continuation is always the record after the line it continues.

LINENUM:

Although the source (EBCDIC - coded value) of the line-number is stored in the workfile, for convenience and efficiency, it is also stored in the STINFO table in decimal form. Any command that requires line numbers can obtain them without accessing the workfile, which would be much slower and more awkward as the workfile source would need to be translated to decimal.

NEXT ST:

This gives the statement number of the next statement in the program. By using the linked list that this sets up, the source can be traversed in the correct order. It is not enough to assume that the next statement will be the one with a statement number one higher, as insertions and deletions will alter this pattern or require immediate reorganisation of the STINFO table. Using a linked-list enables changes to the source to be handled with the minimum of effort, but still makes access to the program relatively easy. The main problem with the linked-lists is that to find

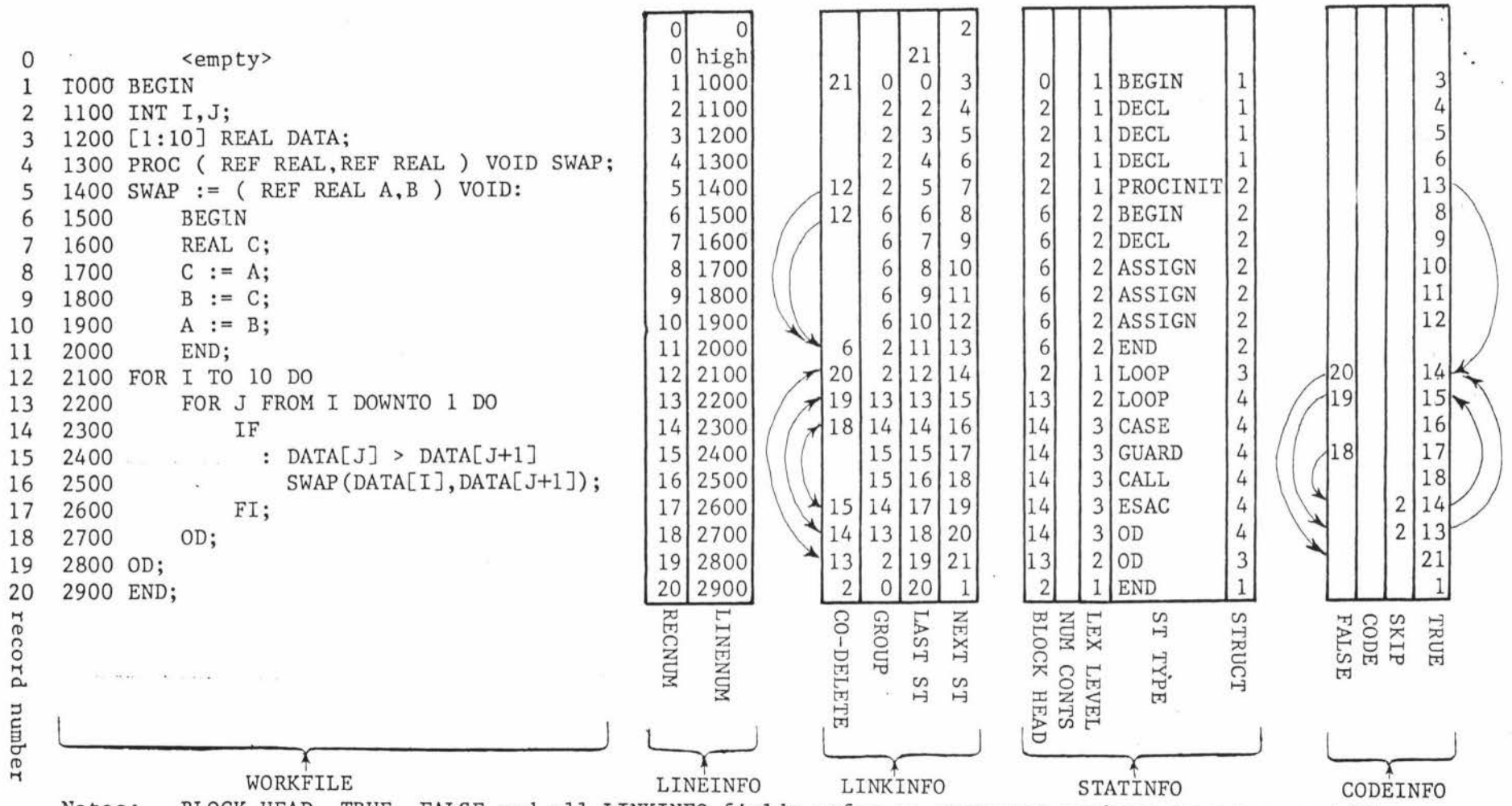


Fig 3.1 STINFO tables for example program.

Notes: BLOCK HEAD, TRUE, FALSE and all LINKINFO fields refer to statement numbers on extreme right.

Only links which are unusual in some way are shown by arrows as well as indices.

Fields are grouped by the way they are organised in the program.

The program, which is a sorting algorithm, contains deliberate bugs.

a statement with a certain line-number, the whole list must be traversed until the line-number is passed. A linear search of this kind is inefficient, especially when it is considered that line-numbers are sequential along the list.

LAST ST:

This gives the statement number of the previous statement in the programs and makes the program linked both forwards and backwards. The NEXT ST of the LAST ST of a statement number is always that statement number. Although it is not strictly necessary, LAST ST makes the program much easier to write.

BLOCK HEAD and GROUP:

These fields are very similar. BLOCK HEAD is a pointer to the first statement of the block that the statement is in. It is mainly used to prevent a user from continuing a program into a block which has not been entered. For this reason, the closing statement of a block (e.g. END or OD) is considered to be part of the block as its trap will occur before the block ends. In STILL, the statement a BLOCK HEAD field points to must be a BEGIN, a loop statement or a procedure initialisation.

The GROUP field has two differences from BLOCK HEAD because of different usage. GROUP is used mainly to prevent deletions which would leave a program syntactically invalid. In addition to the statement types BLOCK HEAD can point to, GROUP can also point to a CASE statement. Also, since the source rather than the code is used in deleting, the closing statement of a group is not considered as part of the group.

CO-DELETE and PREV GUARD:

The CO-DELETE field is another aid to prevent deletions from leaving a program syntactically incorrect. If a statement is to be deleted, then the statement pointed to by its CO-DELETE field must also be deleted. CO-DELETE fields are not active for all statements, and usually come in matching pairs. Thus a BEGIN and an END will have CO-DELETE pointing to each other as will CASE and ESAC and LOOP and OD. The exception to the pairing is when the body of a routine in a procedure assignment is a multiple statement. The last line of the routine and the procedure initialisation statement must have CO-DELETE fields pointing to each other, so the CO-DELETE of the first statement in a routine cannot point to that statement (lines 1400,1500 and 2000 of Fig 3.1).

No PREV GUARD fields are whown in Fig. 3.1. This field is active only for guards of CASE statements other than the first and points to the last guard of the CASE statement. Thus this field is always active for ELSE. A 'next guard' is not required as this information is available elsewhere (see FALSE field). PREV GUARD is used to aid the correct positioning of code threads after deletions within a CASE statement. It is clear that PREV GUARD and CO-DELETE can never be active simultaneously. As a result they occupy the same storage, and the statement type of the statement determines which field it actually is.

NUM CONTS:

This is the number of continuation lines there are for the statement.

LEX LEVEL:

The lexical level of the statement. This information can be obtained by other means, but it is easier to obtain it direct from the STINFO tables. It is used mainly by the TRAP command for checking the range of an ON condition.

ST TYPE:

This field indicates the type of the statement. The types used are language dependent, but adapting for different languages is simple. The statement types are those defined in the trap part of the BREAK and TRACE commands together with DECL (declaration), EMPTY (null statement), GUARD (guard in a CASE statement) and PROCINIT (assigning a routine to a procedure variable).

STRUCT:

This is an index to the structure table and defines the scope at the statement. When compilation is to be done at a particular line-number (i.e. a patch), the structure for the compiler is set up using the STRUCT field of the statement after the insertion.

CODE:

This field is used by the machine and it gives the address of the code for that statement. How the CODE field is used in the code threading is described in Section 3.2.

TRUE and FALSE:

These fields are the threads of the threaded code system. They point to the next statement that should be executed, depending on the value the current statement returns. Most statements return true and just go to the next statement. Loop controls and guards, however, can dynamically return true or false. A guard returns the value resulting from executing the guard, and the loop control is similar, true indicating that the loop has not finished. With a loop control, the FALSE field allows the loop to be exited. The FALSE field of a guard points to the next guard or the end of the CASE statement.

The statement before an OD has a static result, always returning true, but the TRUE field points to the loop control rather than the next statement. This is how loops are implemented.

Statements immediately before guards always return true, but their TRUE fields all point to the ESAC statement closing their CASE statement.

The final special case static result statement is the procedure initialisation. The TRUE field for a statement of this type points to the first statement after the routine. This prevents the routine from being executed at initialisation. The last statement of a routine has no TRUE or FALSE field as before a value could be returned, the routine should return to its invocation point.

SKIP:

This allows the code to jump to somewhere other than the beginning of the next statement. At the moment, SKIP is only used for returning to a loop control after its initialisation.

The next set of fields are the trap information fields not shown in Fig. 3.1.

TRAP:

This is a boolean field and indicates whether a trap has been set at the statement. The other fields are relevant only if this field is true.

BREAK:

This is another boolean field. It is true if the trap was set by the BREAK command and FALSE if it was set by the TRACE command.

CONTROLLED and CONINDEX:

CONTROLLED is a boolean field, and if it is true, the trap is controlled and the value of CONINDEX is relevant. CONINDEX points to the control which controls the trap at the statement concerned. See Section 3.1.4 for a description of controls.

ONCONTROLLED and ONCODE:

ONCONTROLLED is a boolean field which, if true, indicates that an ON condition is attached to the trap. If this is so, ONCODE contains the address of the code for the condition.

HASST and STCODE:

HASST is a boolean field which if true, indicates that the trap has a statement attached to it. If HASST is true, STCODE contains the address of the code for the statement.

3.1.2 Symbol Table

Most of the information in the symbol table is language dependent. Most of the information, however, is used only by the compiler. BIAS uses only the parts that link the symbol table to other data structures.

The symbol table is structured, and has a display, TABLEDIS, to show the current environment (Fig. 3.2). For any environment, the first entry is a marker which contains special information for building the program structure. These fields are:

ENVPTR:

Whereas an ordinary entry would store the address for the variable, a block marker has no address, and this field can overwrite the address. ENVPTR contains an index to the structure table, so that the structure can be obtained from the symbol table if necessary.

TOTALOFF:

This is used to contain the next offset for the block. Addressing is done by address couples (see Section 3.2), and since the lexical level is fixed for any block, TOTALOFF effectively contains the next available address for the block. This is needed if declarations are to be inserted.

```
1000 BEGIN
1100 INT I,J;
1200 [1:10] REAL DATA;
1300 PROC ( REF REAL,REF REAL ) VOID SWAP;
1400 SWAP := ( REF REAL A,B ) VOID:
1500      BEGIN
1600      REAL C;
1700      C := A;
1800      B := C;
1900      A := B;
2000      END;
2100 FOR I TO 10 DO
2200      FOR J FROM I DOWNTO 1 DO
2300           IF
2400             : DATA[J] > DATA[J+1]
2500               SWAP ( DATA[I],DATA[J+1];
2600           FI;
2700      OD;
2800 OD;
2900 END;
```

WORKFILE

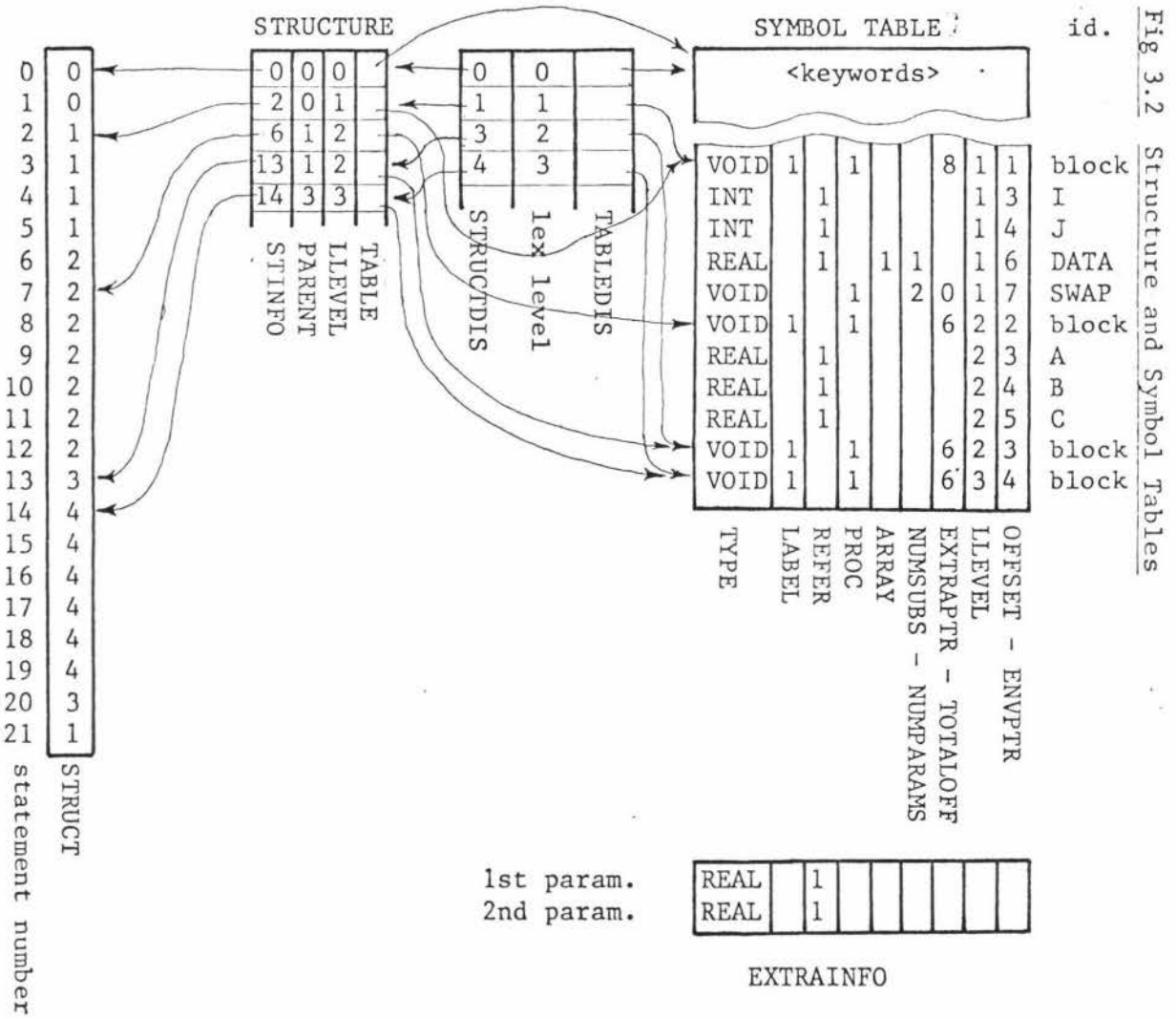


Fig 3.2 Structure and Symbol Tables

Note: The displays are set up for a snapshot at line 2500.

For this implementation, a binary tree system for storing and locating variables was used, as it is fairly fast for insertion and location. Scope rules are handled by first searching at the highest lexical levelled block, (innermost block), and then searching at each lower level until the identifier is found. Reserved words or keywords are in the outermost block, which is also the environment for the program. STILL and BIAS both use keywords and so search for these after checking to see if an identifier was declared. With a reserved word system, this level would be searched first. The fields required for the STILL compiler are:

COUPLE:

This has two parts - the lexical level LLEVEL and the offset OFFSET. This gives the address for the identifier.

EXTRAPTR:

This is a pointer to a special array of additional symbol table information, EXTRAINFO, which is used to describe parameter types. Entries in EXTRAINFO are similar to entries in the symbol table except that the COUPLE field is meaningless in EXTRAINFO. Parameters are fully checked for type at compile-time by using EXTRAINFO.

NUMSUBS & NUMPARAMS:

Only one of these fields can occur at a time, so they are overlayed. If the variable is subscripted, NUMSUBS contains the number of subscripts. If the variable is a procedure, NUMPARAMS is the number of parameters. If NUMPARAMS is non-zero, then EXTRAPTR contains useful information.

TYPE:

TYPE reflects the type of the variable. Simple variables and arrays can be of type int, real or bool. Labels are of type void. Procedures can be either.

PROC, ARRAY & LABEL:

These are boolean fields. If PROC is true, the identifier is a procedure. If LABEL is also true the identifier is a block label or block marker. If ARRAY is true, the identifier is subscripted. PROC and ARRAY are mutually exclusive.

REFER:

REFER is a boolean field. If it is true then the identifier is variable and can be assigned values in assignment statements, read statements and loop controls. An attempt to assign to an identifier whose REFER field is false will give a compile-time error.

3.1.3 Structure Table

The structure table is used to recover the program structure. Each new block has an entry in the structure table which can be seen from Fig. 3.2. Like the symbol table, the structure table has a display (STRUCTDIS) associated with it to show the current environment. The fields of the structure table are:

STINFO:

This field points to the entry in the STINFO tables of the first statement of the block. It is used to check for scope violations in BREAK and CONTINUE commands.

PARENT:

This points to the entry on the structure table of the environment just prior to the new block being entered. By chaining down the PARENT fields, the entire environment for the block can be obtained. If the displays STRUCTDIS and TABLEDIS are to be altered at a breakpoint because the environment has changed, the correct environment is restored to them with these chains. The chains of PARENT fields resemble the dynamic ENV fields of the pseudomachine (Section 3.2).

LLEVEL

This is the lexical level of the block described.

TABLE:

This points to the block marker in the symbol table corresponding to the block described in the structure table. TABLE is used in setting up the TABLEDIS. If STRUCTDIS of level I exists, then TABLE of structure table [STRUCTDIS [I]] = TABLEDIS[I].

3.1.4 Controls

The controls are accessed by the pseudomachine when a controlled trap is executed. Each control consists of four words.

CONTROLCNT:

CONTROLCNT is a count of the number of statements controlled by a particular control. Each time a statement that uses a control has its trap removed for some reason, this count is decremented by one. This enables controls to be freed for further use when no statements are attached to them.

CONTROLBY:

CONTROLBY contains the value of the BY-part of a control in a trap command. CONTROLBY is unaltered while the control remains in use.

CONTROLBYLEFT:

This is initially set to the value of CONTROLBY. A trap using this control which normally would pass control to the supervisor, instead decrements CONTROLBYLEFT by one. If the result is not zero, then the user's program continues running. If the result is zero, the supervisor gets control and the value is reset to the value of CONTROLBY again.

CONTROLFORLEFT:

This is initally set to the value of the FOR-part of a control in a trap command. Each time the supervisor is called at a trap using a control, the CONTROLFORLEFT is decremented by one. When this value reaches zero, all traps controlled by the control are reset, and the control is released for further use.

3.2 Pseudo-machine

As mentioned previously, the pseudo-machine is a stack machine based on the Burroughs B6700 [Dora 73]. The main difference is that the pseudo-machine used here is a threaded code machine and the B6700 is conventional in its code-addressing.

The basic architecture of the pseudo-machine can be seen in Fig. 3.3. The memory is a two dimensioned array M, consisting of 256 rows of up to 4096 words. The rows are allocated as they are needed, and de-allocated when they are not. This gives the machine a lot of potential capacity without always requiring it. Overlaying is left to the B6700 operating system. A program with less than 4K words of array storage will require only three rows of memory (12K). Row allocation is determined by the MEMUSED register, which points to the last occupied word of memory.

Two rows of M are permanently allocated. The last row is XCOUNT and it contains the execution counts for the program. If a reference to this row is made, the index is assumed to be a line-number, so a special decoding procedure is called to translate the line-number to a statement number. If no statement has the line-number, the statement after is used. XCOUNT can be considered as part of the STINFO tables, as well as part of the machine.

The first row of M is the run-time stack S. S contains all the structure of the program, particularly the procedure structure. The last occupied word of S is pointed to by the T register (top-of-stack register). This word is called the A register and the one before it (S[T - 1]) is called the B register, although their location changes as T does.

The memory is tagged, enabling the machine to detect major errors in the code. When a machine error occurs, the type of error and the line it occurred at are displayed. The program is then exited to prevent further mistakes, and control is returned to the supervisor. Tags are also useful as they control indirect referencing. An indirect reference word IRW is a pointer to memory. IRWs can point to other IRW's, and, if a value is required from an IRW, the machine will automatically link down on IRW chain until a word with a different tag is found. The tags are as follows:

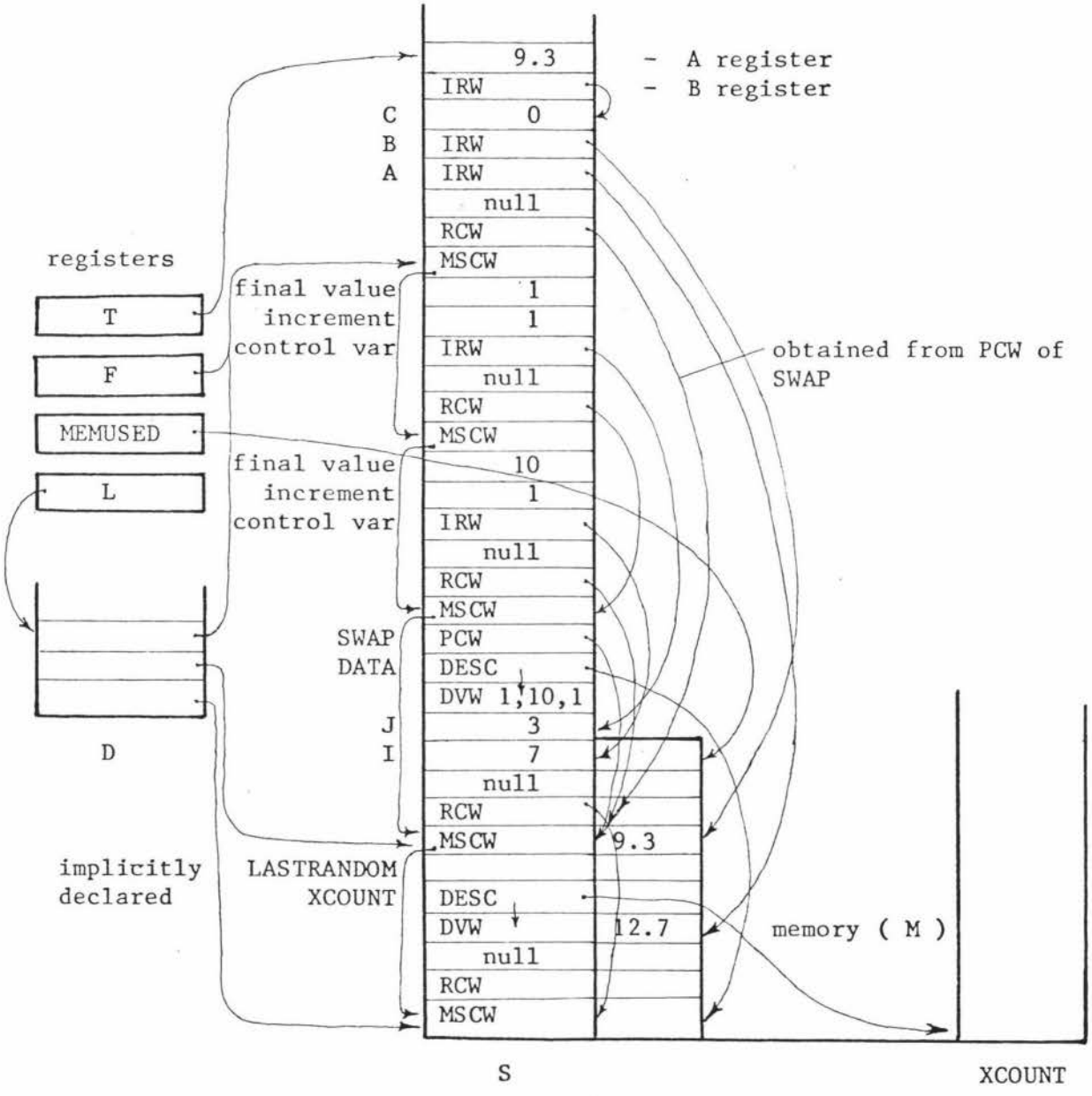
tag	word type	
0 - 7	data word	
8	indirect reference word	- IRW
10	mark stack control word	- MSCW
11	return control word	- RCW
12	array descriptor	- DESC
13	dope vector word	- DVW
14	procedure control word	- PCW

MSCWs and RCWs are used in pairs. The dynamic program links defined by the RCWs yield the display D as can be seen in Fig. 3.3. The lexical level register, L, points to the top of D. The last MSCW is pointed to by the F register. This is normally the same as D[L], but it is required for cases when a procedure is called as a parameter. This is because the MSCW for the parameter procedure will be laid down before the parametered

Fig 3.3 Example showing the design of the pseudomachine.

This is a snapshot of the example program in fig 3.1, taken just before the assignment.

Assume I = 7, J = 3, DATA[7] = 9.3, DATA[4] = 12.7.



procedure is entered, so the environment will not be the parametered procedure, but its environment. This mechanism is described by Doran [Dora 73]. The word on top of a RCW is used to return the value of the procedure or block and is always present though not always used.

All addressing from the code is done by address couples, consisting of the level and the offset. The offset is added to the display of the level to give the absolute address from the base of the stack. Only two instructions use address couples. NAMC pushes an IRW onto the stack, containing the absolute address obtained from the address couple. VALC obtains the word at the address , chaining down IRWs if necessary.

Arrays are described using DVWs and DESCs. A DESC contains the number of dimensions the array has, a pointer to the first DVW for the array and a pointer to the first word of the array. There is a DVW for each dimension the array has. Each DVW holds the upper bound, lower bound and a multiplier for that dimension. Although arrays are stored linearly as in FORTRAN, the last or rightmost subscript varies most, unlike FORTAN. However, a similar indexing algorithm is used. The INDX instruction performs the indexing, pushing an IRW to the appropriate word onto the stack.

The key instruction to threading the code is TRAP. This uses two registers not mentioned so far - the P register, or code pointer and the STAT register or statement-at. When a TRAP instruction is reached, the statement referred to by STAT will have finished. Depending on the value on the top of the stack, STAT will become the TRUE field or FALSE field of the STINFO table entry for STAT, and P will become the CODE field of the new statement modifed by the SKIP field of the statement that just finished. If the statement about to be started is trapped, then this must be resolved. This puts the trap before the statement as was required by BIAS (Section 2.).

If execution counts are required then the count for the new statement is incremented by the TRAP instruction. If the statement is trapped, any ON condition is evaluated first. If there is no ON condition or the condition is true, then the control part is checked (Section 3.1.4). If the trap is still required, any associated statement is executed. Lastly, the trap calls a routine to show how and where the trap occurred, together with an execution count. If the trap was a trace, execution

continues. If it was a break, control passes to the supervisor and the machine is exited. It might be noticed that the pseudo-machine can call itself at the TRAP instruction. Since code emitted by trap commands cannot call procedures, this can only occur to a depth of two, making it safe.

The rest of the instruction set is very straightforward and is described in Appendix D.

3.3. Compiler

Before looking at how BIAS acts on a program, the preparation of the program will be shown. STILL is compiled incrementally, so the user finds most of his syntax errors as soon as he has made them. To take advantage of this, the system must be able to back up one statement at a time.

The STILL compiler was written using recursive descent. This is not an ideal choice for standard complers because of the difficulty of recovery; but with STILL, the major problem, that of finding where to start parsing from after an error, is not present, as the incremental compiler backs up to a known point and starts afresh.

With a structured language, it is essential that the structure cannot be left half finished. This makes it necessary to make editing impossible while a statement is incomplete. Unfortunately, if an identifier was not declared, the omission cannot immediately be rectified. However, since statements can always be inserted afterwards, a statement requiring an undeclared identifier can be omitted, until the declaration can be made.

Most of the program structure is built up by the procedure NEWLINE. NEWLINE always assumes that the program flow will be simple. It links the new statement about to be compiled to the one after it with the TRUE field. NEWLINE also assumes that the new statement will have the same environment as the last and makes the GROUP and BLKHEAD the same. Where these assumptions are false, it is left to the semantic routines and the supervisor to sort the links out. These cases will be shown in detail.

3.3.1 CASE statement

The CASE statement can itself contain several statements and guards. The GROUP field of these must be made to point to the CASE statement itself. After each call on NEWLINE to start a new statement, the GROUP is changed appropriately. At each guard, the PREV GUARD field is set, and the FALSE field of the previous guard is changed to point to the current one. When the ESAC or FI is reached, the TRUE field of each statement preceding a guard is changed to the statement number of the ESAC statement. The final step is to make sure that the CASE and the ESAC point their CO-DELETE fields at each other and their GROUP fields at the same place.

3.3.2 Loop statements

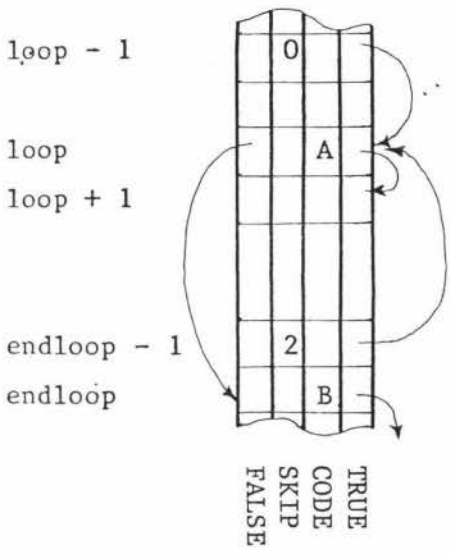
Loops are implicit blocks. Thus on starting a loop, the block entry routine is used to set up the data structure (Section 3.3.3). This automatically causes the GROUP and BLKHEAD fields for the statements within the loop to point to the loop control statement. If an explicit or implicit control variable is present, then the values for the control variable, the increment and the final value are pushed onto the stack as if they were parameters (see Fig. 3.3). The TOTALOFF field in the symbol table for a loop control is thus six rather than three as for a normal block without declarations.

When the OD statement is reached, the CO-DELETE's and GROUP field of the loop control and OD are settled as with CASE and ESAC. The FALSE field of the loop control is pointed to the OD statement, and the TRUE field of the statement before the OD statement is pointed to the loop control with a SKIP field of 2. This is all shown in Fig. 3.4.

3.3.3 Blocks and Procedures

Every block and routine has a new environment associated with it. When the compiler enters a new block, an entry is made in the structure table and the symbol table. Pointers to these entries are pushed onto the displays STRUCTDIS and TABLEDIS. The new structure only is set up as follows. The LLEVEL field is set to the new lexical level. The PARENT field becomes the STRUCTDIS entry under the new top value, which is the environment immediately prior to block entry. STINFO becomes the

Fig 3.4 Code generated for loop statements.



FOR var FROM init BY inc TO final WHILE & cond DO

OD;

```
    A : JUMP                    entry point after statement loop - 1
A + 1 : JUMP                    entry point after statement endloop - 1
        BLCK  endloop,1         enters block storing return address
        <var>                   load address of control var ( NAMC )
        DUPL
        <init>                  load value of initial value
        STOD                    store initial value in control var
        <inc>                   load value of increment
        <final>                 load value of final value
        VALC  level,3           value of control var
        JUMP                    skip incrementing code
        NAMC  level,3           load address of control var
        VALC  level,3           load value of control var
        VALC  level,4           load value of increment
        ADD                     or SUBT if DOWNTO rather than TO
        STON                    store, leaving the value on the stack
        VALC  level,5           load final value
        LEQ                     true while loop continues. GEQ for DOWNTO
        <cond>                  value of the condition
        LAND                    both control and condition must be true
                                to continue
        TRAP                    link to next statement

    B : ENDB                    exit block and go to next instruction
        POP                     first instruction after block, however
                                it is exited. Clears value returned by block
        TRAP                    link to next statement
```

next entry in the STINFO tables, which will be the block start statement. TABLE will be the new entry in the symbol table.

Because every block executed on the machine requires an MSCW, or RCW and a data value to return, the first offset available for use is three. The block mechanism must ensure that each of the fixed words is inserted. Blocks starting with BEGIN are handled differently from procedure body or routine blocks, the former using the BLCK instruction and the latter use MKST and ENTR.

Unless a special case is made, if a typed procedure has a block as a body, no value can be returned to the procedure as two block entries would occur and there would be no way of referencing the outermost one. Thus if a block is a procedure body, a new environment is not created, and the environment within the block is the routine (see Fig. 3.1). In particular this means that declarations made in such a block and formal parameters of the routine are declared at the same level.

When a routine is assigned to a procedure variable, the routine itself must not be executed. All that must be done is to assign a PCW describing the routine to the variable. the PCW is set up in the actual assignment statement, so the rest of the routine must be skipped over. This is done to make the TRUE field of the procedure initialisation statement point to the statement after the last statement of the routine. The last statement of the routine has no TRUE field, as before its TRAP instruction can be reached, it should have returned to the calling point of the procedure.

If a block is assigned as a routine, there are three statements which must be deleted at the same time - the assignment, the BEGIN and the END. This sets up the CO-DELETE fields in a slightly different manner to usual. The assignment and END point to each other, but the BEGIN also points to the END (see Fig. 3.1).

3.4 Command Analyser

The command analyser is closely tied in with the data structures. It has already been mentioned how the data structures were designed to enable the commands to work easily. This has made the system simpler to write and understand.

3.4.1 Trap commands

The parsing of the TRACE and BREAK commands are fairly straightforward. The only unusual part is that the expressions of the BY and FOR parts are evaluated immediately. All references to the prgram assumes the environment to be that of the position in the program where the program is trapped.

Once the command has been parsed, the scope of the command is determined. If there is no ON-condition or the ON-condition uses globals only, the entire program is the scope. For any ON-condition the outermost scope referred to by the condition is the scope. For all statements in which the scope is valid, a boolean field, VALIDSCOPE, is set.

If the traps are to be controlled, an available control is found. The CONTROLBY and CONTROLBYLEFT parts are set to the value of the BY-part of the control which defaults to 1. The CONTROLFORLEFT is set to the value of the FOR-part and defaults to a large real number. This number is large enough so that subracting one from it will leave it unaltered because of rounding.

Each line-num-seq of the trap-part is then linked through using the NEXTST field. No action is taken unless a statement reached by this linking has its VALIDSCOPE field set and is of a statement type included in the trap-part. If the statement is already trapped and not controlled, the CONTROLCNT of that control is decremented by one. The trap information fields can then be set.

The UNBREAK and UNTRACE commands do not have to worry about scope. All they do is to link through the statements indicated, turning off all traps if the type of the statement was included in the trap-part.

The SET and RESET commands flip the flags BREAKSET, TRACESET and XCOUNTSET which are used by the machine to determine if action should be taken at a trap. Resetting XCOUNT also sets all execution counts to zero. Each of these flags defaults to the set state when the system is run.

3.4.2 CONTINUE command

If there is no line-number mentioned by the command, the program just carries on where it left off by entering the pseudo-machine. If a line-number is used, then before execution can continue, the statement

at the line-number must have its environment checked to make sure that its scope is the same as scope of the statement that the program has stopped at, or least global to it. If the scope is the same, the structure on the pseudo-machine can be left unchanged. However, if the scope is global, the run-time stack has to be cut back to the global level. This is why other scopes are not valid; they would require stack building to be done without knowledge of the type of word being placed on the stack or where the calling point for the block or routine is.

3.4.3 DELETE command

The first phase of analysing a DELETE command is to make sure that every line-num-seq is valid. This ensures that deletion is not done if there is any mistake in the syntax as an accidental mistake might otherwise cause much irreparable damage.

The changes to the STINFO tables occur at four statements. These are the first statement to be deleted, FIRSTST, the statement preceding this, STBEFORE, the last statement to be deleted, LASTST, and the statement after this, STAFTER. However, before any changes can be made, a check must be made to ensure that the program will be syntactically valid after the deletion has occurred. Three conditions must be satisfied for this:

1) The group fields of FIRSTST and LASTST must be the same. This prevents most cases of deleting only one end of a compound statement. The exception, i.e. when the first or last statement of a compound statement is LASTST or FIRSTST respectively, is handled by the second condition. This condition does prevent certain deletions which may corrupt the code, although leaving the program syntactically correct. In example 3.1, if lines 7600-7900 were deleted, the program would be syntactically correct, but would cause a pseudo-machine fault at line 8000 as the address couple for B would be invalid. Again in example 3.2, if lines 3800-4000 were deleted, the program would be syntactically correct, but the changes to the STINFO table would be many and similar deletions (e.g. 3900-4200, 3900-4100) would require greatly differing changes, making such deletions very context sensitive.

Example 3.1 STILL

```
7400 BEGIN
7500 REAL B;
7600 B := 2;
7700 END
7800 BEGIN
7900 REAL A,B;
8000 B := 3;
8100 END
```

Example 3.2 STILL

```
3500 A := 1
3600 IF
3700    : B = 2
3800      C := 3;
3900 FI;
4000 IF
4100    : D = 4
4200      E := 5
4300 FI;
```

2) If STARTST or LASTST has a non-empty CO-DELETE field, then the statement referred to by that CO-DELETE must be between FIRSTST and LASTST. This can be easily checked using the line-number of the statements involved. This condition completes the check that if one end of a compound statement is deleted, then the whole statement is deleted. This takes care of such deletions as 3900-4000 in example 3.2. Lines 3900 and 4000 will have the same GROUP field, but the problems mentioned above will still arise. This second condition makes this event illegal.

3) If STBEFORE is a CASE statement, then STAFTER must be a guard. This prevents the deletion of line 5200 and lines 5200-5500 in example 3.3. Both of these would leave the syntax invalid.

Example 3.3 STILL

```
5100 CASE
5200   : A = 1
5300     B := 2
5400   : C = 3
5500     D := 4
5600   ELSE
5700     E := 5
5800 ESAC
```

One semantic condition also applies - declarations cannot be deleted unless their whole scope is too (i.e. the block they are in is also deleted). Checking FIRSTST is all that is required. Since declarations precede statements and immediately follow a BEGIN, if any declarations are deleted, FIRSTST must be a declaration as the syntax conditions prevent it from being anything else.

If the deletion occurs within a CASE statement, which means that the GROUP of FIRSTST refers to a CASE statement it may be necessary to link around the guards. If a guard is deleted, then the last guard before the deletion is found using the PREV GUARD field, and the guard after is found using the FALSE field. These statements then point to each other (see Fig. 3.5 (a)).

All statements to be deleted are the scanned, decrementing the CONTROLCNTs of controls which have traps removed. The TRUE and SKIP fields of STBEFORE become the TRUE and SKIP fields of LASTST. The LAST ST and NEXT ST fields of STBEFORE and STAFTER are changed so that they are logically successive statements (see Fig. 3.5 (b)).

If the CO-DELETE of STBEFORE is a procedure initialisation, then the TRUE field of that statement will point to FIRSTST. A check is made, and if this occurs, the TRUE field of the procedure initialisation is changed to point to STAFTER (see Fig. 3.5 (c)).

3.4.4 Patching

A patch is detected by the line-number which starts it. Most frequently a patch means entry of a statement to overwrite an existing statement or the insertion of a statement between two other statements. It is also possible to overwrite a loop control or a guard with a new loop control or guard.

a) DEL 5400

```
5100 CASE
5200    : A = 1
5300      B := 2;
5400    : C = 3
5500      D := 4;
5600    ELSE
5700      E := 5;
5800 ESAC;
```

b) DEL 2300-2600

```
2200      FOR J FROM I DOWNTO 1 DO
2300          IF
2400              : DATA[J] > DATA[J+1]
2500                SWAP(DATA[I],DATA[J+1]);
2600          FI;
2700      OD;
```

c) DEL 8500-8600

```
8300 FUNCT := ( REAL PAR ) REAL:
8400      RETURN PAR*SIN(PAR);
8500 A := 1;
8600 B := 2;
8700 C := 3;
```

Note: Values before deletions are shown in parentheses at side.

All changes are marked by circles.

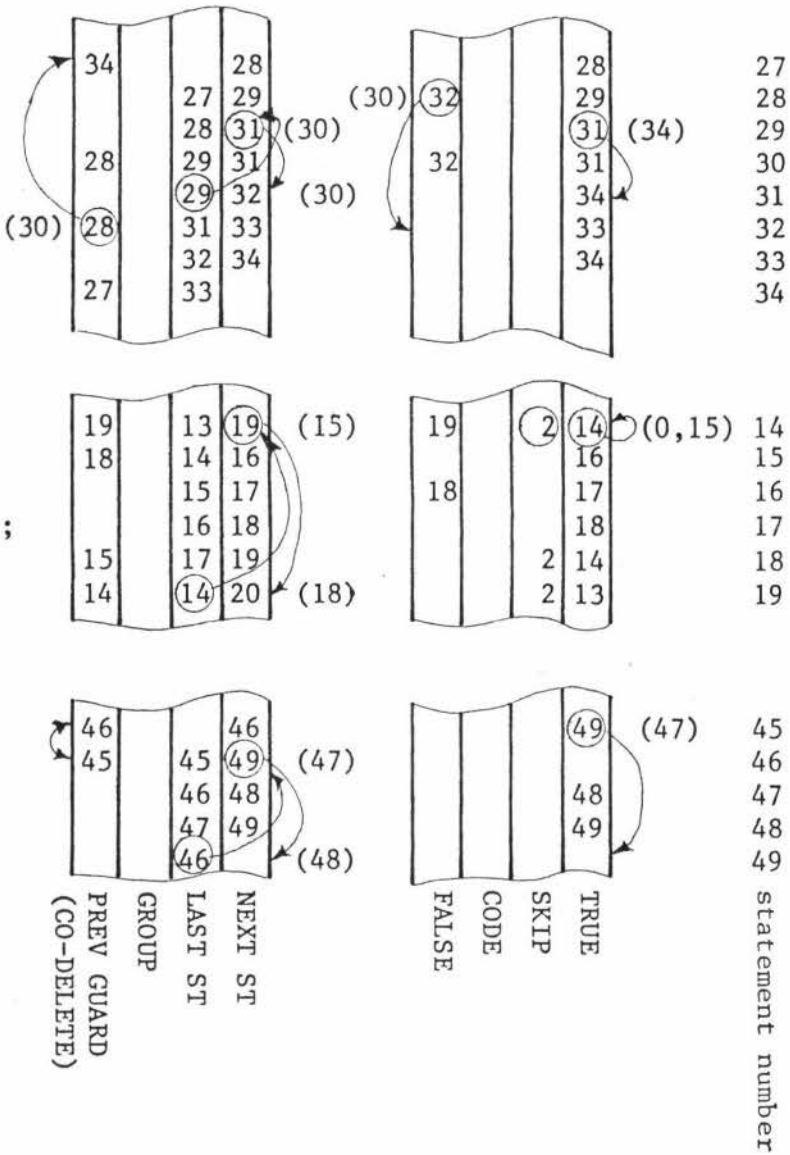


Fig 3.5 Examples of deletions

In all cases, the first thing that must be done is to set up the compiler displays so that the structure is correct for where the patch will be made. If the new line is an overwrite, the type of the statement it is overwriting is checked. If it is a guard or a loop control, the new line is considered to be the same. The source is entered at the end of the workfile and the code is emitted at the end of the code array. The RECNUM and CODE fields are the only fields for the line that are altered (Fig. 3.6 (a)). For all other overwrites, the statement must be a simple statement, and the patch is similar to an insertion.

Insertions start by noting the statements before and after the insertion STBEFORE and STAFTER. The new statement is compiled as if it directly followed STBEFORE, and the first line of the patch is set up as if NEWLINE had been called. If the line-number of the last line of the patch exceeds the line-number of STAFTER, it is an error, even though this may have been desired. To overwrite more than one line, the lines must be explicitly deleted first.

So far, the patch links back to STBEFORE. The logical links, LAST ST and NEXT ST are then completed so that the sequence of the lines is correct. The code links are harder to fill in, and differ for overwrites and insertions. The TRUE and SKIP fields of the last line in the patch are set to those of STBEFORE if the patch is an insertion or those of the line that was overwritten if the patch is an overwrite. The TRUE field of STBEFORE is pointed to the first line of the patch. No FALSE fields can be affected by insertion. If the CO-DELETE of STBEFORE is a procedure initialisation, the TRUE field of that statement is pointed to the first line of the patch and the TRUE field of STBEFORE is made null. This makes sure that the patch will be executed immediately after the procedure assignment. Examples of an overwrite and an insertion can be found in Fig. 3.6 (b) and (c) respectively.

3.4.5 Other Editing commands

The file handling commands, GET, MAKE, SAVE and REMOVE, are mostly devoted to ensuring that B6700 I/O subsystem requirements are met. These conditions of the workfile are used and may be set: there is a workfile, the workfile is empty and the workfile is saved.

Fig 3.6 Examples of insertions

a) line 2100 overwritten

```
2000      END;                          11
2100 FOR I TO 10 DO                     12
2200      FOR J FROM I DOWNTO 1 DO      13
```

b) line 7400 overwritten

```
7500 FOR I TO 10 DO                     69
7600      S := S + A[I];                70
7700 OD;                                71

7600      S2 := S2 + A[I]*A[I];         97
```

c) line 5350 inserted

```
5100 CASE                               61
5200    : A = 1                         62
5300      B := 2;                       63
5400    : C = 3                         64
5500      D := 4;                       65
5600 ESAC;                              66

5350      E := 5;                       82
```

Note: Values before insertion are shown in parentheses st side

All changes are marked by circles

The LIST command is set up to link through the line-num-seqs given by using the NEXT ST links in the STINFO table. At each statement linked through, the record at RECNUM for that statement is listed, the workfile being randomly accessed. If NUMCONTS is not zero, then that number of consecutive records are then listed before linking to the next statement. To prevent interactive users getting caught watching long listings go by, only enough lines for a page will be listed before listing will suspend and the user is asked for input. If he enters a null line, listing continues. If the line is not null, it is treated as a command.

The RUN command clears the pseudo-machine and sets the registers to the correct values for running the program from the start. It then calls up the pseudo-machine, which runs until the program finishes or a break-point is reached.

3.4.6 Immediate commands

Immediate commands are source language statements which are compiled and executed immediately. When the compiler is called, a flag is set to indicate that code generated will not be permanent. This prevents the source of the command from being written to the workfile.

Immediate commands require special pseudo-machine instructions to start and finish the code. The code generated is treated as a block. but unlike other blocks which return to the statement after the block, immediate commands have no statement after them. The STRT instruction sets up the code so that it thinks it was called from the position where the program has stopped running. The FNSH instruction is like an END but it also causes the pseudo-machine to pass control to the supervisor.

3.5 Linkage

It is usual for the debugging routines to be part of a procedure which the machine calls whenever it reaches a breakpoint. However, if procedures of the user's program can be called successively without waiting for them to be exited, this can cause problems to the user when he finally does start backing out of them. He will find it difficult to know which routine he is currently in. For this reason, BIAS does the opposite. The supervisor calls the pseudo-machine. This has the added

advantage of requiring much less memory, because the system can only nest itself to a level of two.

However, the supervisor does not know when the pseudo-machine has run. This means that the supervisor and the user's program are run more as co-routines than as calling routine and procedure. In fact, this could have been done very easily on the B6700, but although this would have provided a more realistic model of the actual structure, a much larger amount of real resources would have been required.

Chapter 4

USING THE SYSTEM

"He is very fond of making things he doesn't want,
and then giving them to people who have no use for them."

The Dolly Dialogues Anthony Hope

The best way to show how a system is used, is to show it in action. For this reason, an example program which contains deliberate bugs is followed through its various stages until it runs correctly. The program used is the program of Fig. 3.1 which has already been used throughout Chapter three.

4.1 Entering the Program

When a user starts running BIAS, he will receive a prompt from the system which will be a hash-mark, "#". This signals that the system is ready to receive input. All commands given by the user are in response to some prompt. Throughout this chapter, all text written by the system will be underlined.

The first command given by a user will normally be a GET or MAKE. In example 4.1 the user makes a STILL file called SORT. At this stage this file does not actually exist and any attempt to use it for anything except source insertion will cause an error.

After the make commands, a hash-mark is prompted again. This time the user enters the first line of text in the program. BIAS now recognises that a statement has been started and calls the STILL compiler. Until the statement is finished, all prompts for text will be colons. When the user enters the final line of the program, the next prompt will be a hash-mark. This helps him know that the compiler has recognised the line just entered as the last line to be entered and that a command is expected.

Example 4.1

```
#MAKE SORT
#1000 BEGIN
:1100 INT I,J;
:1200 [1:10] REAL DATA;
        :                          } as for Fig. 3.1
        :

:2700     OD;
:2800 OD;
:2900 END;
#2020 WRITE SAME "?":A1;
#2030 READ DATA[1]:I3;
#2040 FOR I FROM 2 TO 10 DO
:2050     READ SAME:T I*3,DAYA[I]:I3;
                                *<<<< UNDECLARED IDENTIFIER
#LAST CORRECT STATEMENT AT 2040
:2050     READ SAME :T I*3-2,DATA[I]:I3;
:2060 OD;
#2840 FOR I BY 2 TO 10 DO
:2850     WRITE DATA I :I5,DATA[I+1]:I5;
:2860 OD;
```

Although BIAS and the STILL compiler prompt the user for input, the pseudo-machine does not. However, the user can easily arrange for prompts by using a statement like line 2020. Note that the string "?" having four or less characters is treated as an integer, as so needs to be formatted to give the required result.

When the user first entered line 2050, he made a small typographical error. When the compiler discoverd this it informed him of both the nature of the error and where the compiler has backed up to. He now knows that he should continue entering his patch at line 2050.

4.2 Run-time Errors

Having entered the program and thus compiled it, the next stage is to run the program. This is done with the RUN command. At first the program runs correctly; the user is prompted by his program and he enters his data (Example 4.2). Unfortunately, before his program can

provide any output, an array subscript exceeds the upper bound of the array. This causes the pseudo-machine to give a run-time error.

Example 4.2

```
#RUN
? 42 67 41 29 02 92 23 88 42 19
SUBSCRIPT OVERFLOW AT 2400 (*7)
#WRITE I,J
   10   10
#2100 FOR I TO 9 00
#J := 9;
#CONT
   42   67
   67   67
   67   92
   92   92
   92   92
#WRITE XCOUNT[2500]
    7
```

Subscript overflow does not cause the program to be aborted. The code is backed up to the start of the statement, and the user can then change values which will prevent the error. As far as the user is concerned, a breakpoint has just occurred. In example 4.2, he checks the values of I and J and realises that there is a mistake in the final value of the outer loop. He then inserts the patch to correct this. J, however, is still 10, so he changes this value with an immediate command, and the new loop control will prevent it from becoming 10 again. The user then continues his program at the point it left off. The code that caused the fault will be executed again, but should not cause a fault this time.

Instead of changing J and continuing from the point of error, the user could have skipped over the false code with the command CONT 2700. This would cause the inner loop to be exited, which is what should have happened anyway.

4.3 Module Testing

After the program resumes execution, the user gets some output but the values are completely wrong (Example 4.2). He looks at the

execution count for the invocation of the procedure SWAP (line 2500). This shows that the number of values which were erased is the same as the number of times SWAP was called, and leads him to suspect SWAP as the cause of the error.

The best way of testing SWAP is to call it using known values as input. To be able to do this, the program must be stopped at some point that contains SWAP in its scope at a time when the routine has been assigned. The user inserts a breakpoint at line 2100, limiting it to the one occurrence with a control (Example 4.3). He then runs the program, re-entering the data.

Example 4.3

```
#BREAK 2100 FOR 1
#RUN
? 42 67 41 29 02 92 23 88 42 19
BREAK @ 2100 (* 1 )
#WRITE DATA[1]:I3,DATA[2]:I3;
 42 67
#SWAP (DATA[1],DATA[2])
#WRITE DATA[1]:I3,DATA[2]:I3;
 42 42
#DATA[2]:= 67
```

When the breakpoint is reached, the user is told what line he is stopped at and given the execution count for the statement. Since SWAP requires ref real parameters, declared variables will have to be given, so the values of these, DATA[1] and DATA [2], are checked first. The procedure SWAP can then be called. This method allows a user to call any procedure in his program without having written any other parts of the program except for inserting the globals required by that procedure. If a procedure which has not had a routine assigned to it is called, a breakpoint occurs. The user can then cause all the effects that the missing procedure could do if it were present and correct.

After SWAP has been executed, the user checks the values returned by the parameters. It is now plain that there is a bug in SWAP, but before he discovers what it is, he repairs the damage to the data.

4.4. Editing

The cause of the bug is soon apparent; statements 1800 and 1900 are back to front. This is fixed by deleting one of these lines and then inserting that line at the correct place (Example 4.4 (a)). The program is then continued from the breakpoint (at 2100). If, however, the user had decided to completely rewrite the routine and its assignment to SWAP, then SWAP would still have the incorrect routine, as the new, correct source would not have been executed and the new routine assigned (Example 4.4 (b)). This method would require the assignment to be done by some means. By continuing at line 1400, the procedure initialisation, this problem is solved, but the data must be re-entered.

Example 4.4.

(a)
```
#DEL 1900
#1750     A := B;
#CONT
```

(b)
```
#DEL 1400-2000
#1400 SWAP := (REF REAL A,B) VOID:
:1500      BEGIN
:1600      REAL C = A;
:1700      A := B;
:1800      B := C;
:1900      END;
#CONT 1400
? 42 67 41 29 02 92 23 88 42 19
```

The SWAP routine is now correct, and the program produces output again, and again the output is wrong (Example 4.5). By this stage, the source text of the program will probably have been lost off the top of the terminal screen, so the user lists the main part of the program.

Example 4.5

```
  42  02
  41  67
  88  23
  42  19
  29  92
#LIST 2100-2800
2100 FOR I to 9 DO
2200     FOR J FROM I DOWN TO 1
2300         IF
2400           : DATA[J] > DATA[J+1]
2500             SWAP (DATA [I], DATA [J+1]);
2600         FI;
2700     OD;
2800 OD;
```

4.5 Breakpoints

The user now has to find out where his program is going wrong. He decides to dump the values of some of his variables at a suitable point in the program. Since he will want to stop the program when he sees something go wrong, he uses a BREAK command rather than a TRACE command (Example 4.6(a)). The dump is done by attaching a WRITE statement to the trap. If all the elements of the array had been required, then the command of Example 4.6 (b) would be used. Note that in this case, an indexing variable has to be declared as there are no variables in the program that could be used in the loop control. Example 4.6 (b) also shows how the name of the variable can be made to appear with the output. Naturally the more information displayed, the more lines of output used, and this may obliterate information that is still wanted. For this reason, BIAS keeps the number of lines it outputs to a minimum, and a user will often do this as well.

Example 4.6

(a)
```
#BREAK 2600:WRITE I,J,DATA[1],DATA[2],DATA[3],DATA[4]
#RUN
? 42 67 41 29 02 92 88 42 19
     1     1          42          67          41          29
BREAK @ 2600 (* 1 )
#                              note: this is a null line
     2     2          42          41          67          29
BREAK @ 2600 (* 2 )
#
     2     1          42          41          67          20
BREAK @ 2600 (* 3 )
#2400                SWAP ( DATA [J], DATA[J+1] )
#UNBREAK
#SWAP ( DATA[1], DATA[2] )
#C
    2   19
   23   29
   41   42
   42   67
   88   92
```

(b)
```
#BREAK 2600:BEGIN
:INT K;
:WRITE "I = ", I," J = ", J;
:FOR K BY 2 TO 10 DO
:    WRITE K,DATA[K]:I5,K+1,DATA[K+1]:I5
:OD
:END
```

The program is run again and the data re-entered (Example 4.6 (a)). The first and second breakpoints reached show nothing amiss, so the user continues his program by entering a null line, i.e. one with no characters in it. This is only allowed as a continuation symbol at the first reponse to a breakpoint. If a null line is entered at any other time, it is treated as a blank line.

The next time the breakpoint is reached the user notices that no elements of DATA bave been interchanged when the first and second elements

should have been. He looks at the call on SWAP and realises that he has used DATA [I], instead of DATA [J]. He then inserts the patch to correct the error. The breakpoint is removed, the data is restored to the way it should be and the program is continued. The program outputs the results which are now correct.

4.6 Finishing Up

The final stage in a session will involve manipulating the workfile. If the program is not required, the user may just remove it by the command REMOVE. He can also save it, changing its title if he wants to. This is often useful if the workfile was obtained by a GET command, but the original version is required for back-up. Since a SAVE command without a file-title would cause the original to be overwritten, an alternative must be available. Saving with a file-title will create a file of that title and save the workfile in it.

In example 4.7, the user tries to save his program under a different title. However, he already has a file with that title, and to prevent users from overwriting any file except the workfile, BIAS reports an error. The user decides that he does not want the old file, so he removes it and does his SAVE command again. Finally he leaves BIAS by using the QUIT command. Since his workfile is saved, the system will be exited. This is not the case if the user has an unsaved workfile, and an error will result if he tries to exit the system while this is so.

Example 4.7

```
#SAVE SORTING/PROGRAM
                        *<<<<  FILE ALREADY EXISTS
#COMMAND IGNORED
#REMOVE SORTING/PROGRAM
#SAVE SORTING/PROGRAM
#QUIT
```

4.7 Running under Batch

Although incremental compilers are made primarily for interactive use, they can also be used in batch. When running in batch, it is best to start by just compiling the program and not to execute it

until it compiles correctly. However, BIAS does not require this and can run every program as syntactically incorrect statements will just be ignored. A preliminary run on BIAS in batch would normally consist of a MAKE command, the program text, a SAVE command and a QUIT command. Subsequently a GET would be used to retrieve the program, and patches would be merged. As stated earlier (Section 2.4), this does not entail a complete recompilation; only the patches would be compiled just as they would if they were entered interactively.

After the new source for a run of the system has been entered, various commands to control the running of the program will be entered. Dumps and traces will be entered as they would be from a terminal and the program would be run. These may well be followed by contingency commands to handle machine faults. If the program ends normally, these may be ignored or they may be in error, but this will not affect the actual running of the program.

It can be seen that running BIAS under batch is not very different from running it interactively, and the same tools are available for both. The main difference between the two methods will be the debugging strategy. Batch users will depend heavily on the TRACE command and will use large-scale dumps so that they can be sure of having the right information to track down any bugs. Interactive users will mostly use the BREAK command, and dump only the minimum required to find any bugs.

Chapter 5

CONCLUSIONS

".... he was not quite sure what to do next.

But he would think of something."

2001, A Space Odyssey Arthur C. Clarke

Before making any conclusions about BIAS, it will be useful to see how well it fulfils its three major design goals (Chapter 2). Simplicity was certainly achieved. The only noticeable defect was that the commands to display the values of variables were quite long-winded. This, however, is basically a problem with STILL, and although BIAS could have contained a command for this purpose, language independence would have been lost.

Four types of generality were required. Program independence was achieved as BIAS imposed no restrictions on programs except in its format requirements. These do not restrict the source of the program only the way it is set out. Although BIAS is primarily interactive, it requires no changes to run under batch, and the same debugging tools were available for both.

Language independence has not been proven as only one language has so far been implemented for BIAS. However, there seems to be no reason why other languages could not be used under the system. BASIC would be particularly easy to implement as it is designed to be incremental. FORTRAN could be used as a host language without much trouble. The main difficulty would be a way of relating the FORTRAN statement labels to the line-numbers of the source. There would also be very little difficulty in fitting COBOL onto such a system, although it would take a lot of work because of the size of the language. The case of structured languages such as Algol and PL/I has been demonstrated with STILL. Only formatting restrictions would be imposed as they were for STILL, but the basic method of handling structured programs still applies.

Machine independence has been mentioned before (Section 2.4). Since the pseudo-machine can be a program in its own right, it would be possible to run BIAS on it. This only requires a host language on BIAS suitable for writing the software and an implementation of the pseudo-machine. As

mentioned earlier, the pseudo-machine is short enough and simple enough to be written wherever BIAS is to be used.

The last design goal was to make the system incremental. This is achieved by using threaded code and requiring one statement per line. It can be seen that the cost of making the system incremental is restrictions on the format of user programs. However, it is felt that the power and case of use obtained by having a fully increment system far outweights restrictions which good programming style would enforce anyway.

Implementation of BIAS is not complete as yet. There are still several bugs in the trap commands which are only revealing themelves as the system is tested extensively. There is no garbage collection routine as yet, and store tracing has not been implemented, although these are in the planning stage. All of the other BIAS commands appear to work correctly, although with breakpoints not working correctly it is impossible to tell whether all immediate commands will work correctly when applied to data in a program. However, they are known to work outside the scope of the program.

Several extensions have been planned. It is hoped to extend STILL to include string handling, data structures and more powerful I/O. This would involve adding instructions to the pseudo-machine, taking care that any instruction will not be too STILL specific.

There is a lack of editing facilities in BIAS. There could be at least a fix command which would change a part of a line and resubmit it to compilation. An automatic sequence made would also be of great use as it can be tedious to have to enter all the line numbers. The ability to resequence the source would also be useful for cases where a patch is required between two lines with consecutive numbers. A cross reference analyser would also be useful.

Another useful debugging feature would be the ability to reverse the execution of a program to see how it reached a certain state. However, this would be extremely difficult to implement, although the benefit would be great.

As it stands, BIAS has not had sufficient use to guage its usefulness. To evaluate its merit properly would require its being used by users of

different types and experience running programs of varying size and complexity. Since BIAS has not been completely tested yet, only the author has had experience with it, but its advantages have still come through, and interest has been expressed by several other programmers.

BIAS is of particular use at Massey as until very recently there was no incremental compiler available for use. It certainly has a big advantage against conventional compilers for testing small algorithms as BIAS only ever requires to be run once, whereas other systems require a run for each compilation and another to execute the program each time a correction is made. When testing the program of Fig. 3.1, the B6700 Algol version underwent the same debugging procedures. One of the three bugs in the program was unknown when the program was entered. This was found, and other bugs corrected in less than half the elapsed time and much less machine time using BIAS than just correcting all the bugs with forknowledge of them using B6700 Algol. This was done without using traps. Of course, the B6700 Algol version will run a lot faster than the BIAS version, although this difference is significantly reduced if B6700 Algol breakpoints are used. Programs under BIAS run at the same speed regardless of whether any breakpoints are set.

Comparing the breakpoint facilities of BIAS and B6700 Algol puts BIAS ahead for the most part. The patching and immediate mode facilities of BIAS are almost completely lacking in B6700 Algol, although displaying variables is easier with B6700 Algol. A B6700 Algol capability which is long winded in BIAS is the ability to continue until a given line is reached. BIAS has to do this by

```
BREAK line FOR 1
CONTINUE
```

whereas B6700 Algol uses the much simpler one-line command

```
/CONTINUE line
```

Considering that "CONTINUE line" has a different meaning in BIAS, it cannot use just one line. However, if "FOR 1" was the default, the difference between the two would be minimal. Whether or not any of the defaults are the right ones can only be found out from experience.

BIAS was originally conceived as a debugging system which included the ability to make permanent patches to the program at run-time. It has however, shifted its stance to being more of an incremental compiler host containing debugging commands. This tends to show that the two are really different manifestations of the same beast. It has long been known that incremental compilers aid debugging, but surprising little has been done to extend them to contain complete debugging systems. Likewise, most debugging systems do not have the ability to make permanent patches to the source program. Of course, this is extremely difficult to do without completely rewriting much existing software, but there are great benefits to be gained.

This does show the advantages of a threaded-code machine. If such a machine were implemented in hardware, it would greatly aid the development and use of debugging systems. The increase in machine-time for running a program on such a machine as against conventional machines is very small. Also, as it is used in conjunction with the stack machine as was done in BIAS, the size of the code is much shorter than for conventional machines.

Appendix A

Comparison of Debugging Systems

System	How implemented	Patching	Language
1 MANTIS Ashb 73	interactive standard code	assignments and I/O cannot patch patches	FORTRAN
2 EXDAMS Balz 69	interactive precompiler	none	any
3 Bate 74	interactive interprets pseudocode	none	-
4 PEBUG Blai 71	batch/interactive compiles executes or interprets code	none	-
5 Bull 72	interactive incremental compiler threaded code	full language as incremental compiler is used	BASIC
6 Burr 76	interactive compiler inserted breakpoints	none	Burroughs Extended Algol
7 BUGTRAN Ferg 63	batch precompiler	none	FORTRAN
8 CONTROL Gain 69	interactive compiles physical breakpoints	none	FORTRAN (and others?)
9 AIDS Gris 70	interactive compiles executes or interprets code	IFs,assignments,PRINT and GOTOs. cannot patch patches	FORTRAN assembler
10 Grov 74	batch interprets traps built into machine	none	MUSSEL
11 HELPER Kuls 71	interactive compiles executes or interprets code	IFs,GOTOs,assignments cannot patch patches	FORTRAN assembler and others
12 DDS Pier 74	interactive compiles	large language subset cannot patch patches	Coral
13 Ryan 66	interactive incremental compiler	full language as incremental compiler is used	Algol 60 FORTRAN
14 Satt 72	batch compiles	none	Algol W
15 MULTICS Wolm 72	interactive compiles physical breakpoints	bind in procedure	PL/I
16 BIAS	batch/interactive interprets threaded code	full language incremental compiler is used	STILL (and others)

	dumps	traps and traces	controls	variables
1	selective using PRINTS with ATs	store,call,return line	GO TO	examine store consts
2	not as such	not as such	none	examine only
3	none	line and store can be suspended	simple conditional jumps	examine store consts
4	selective at trap At machine level	overlay,store,call flow,error,every instr,any instr	simple conditions go to anywhere	examine store consts or vars
5	indirectly using incremental compiler	line trace (every line)	implicit with incremental compiler	examine and store full expressions
6	selective if compiled in permanent	store,label if compiled in permanent	none	examine most store consts to simple var
7	selective	flow, call and store in range	full expressions DO loops	no
8	selective	line only	none	examine store consts
9	snap of registers selective using PRINT with TRAP	opcodes,line load,store,call	IF expression without calls GOTO anywhere	examine and store (no proc calls)
10	selective and post-mortem	store,fetch, flow,call,source	asserts	no
11	registers,core single vars	line,store,load, opcode,flow	IFs,GOTOs and expressions	examine and store
12	none	line only	none	examine store consts
13	indirectly using incremental compiler	none	immediate mode language	examine and store full expressions
14	selective and post-mortem	store,call, conditions, statements	asserts	no
15	-	-	-	examine and store
16	selective formattable	line,store, statement type	loop controls and conditions at trap immediate mode	examine and store full expressions

	stepping	reversing	remarks
1	no	no	At source level Allows user to cross scope No abbreviations
2	yes	yes to any degree	Very unconventional Shows complete execution of program dynamically
3	no	yes	At source level. No abbreviations Can cross scope. Can back up through procedure calls
4	indirect	yes, by checkpoints	Unnatural abbreviations Drops to machine level in places
5	no	no	No abbreviations Very natural and easy to understand
6	basic mode	no	Very restriced Requires control characters for different modes
7	no	no	Very clear and easy to understand
8	yes	yes by checkpoints	Requires a specific type of terminal Very machine oriented
9	yes	yes	Very wordy and occasionally confusing Has machine level constructs
10	no	no	Has execution counts Good control of debugging facilities
11	yes	yes	Wordy and occasionally confusing Has machine level constructs Compiles commands
12	yes	no	Abbreviations only (one letter) Mainly for editing on a mini.
13	yes	no	Good abbreviations At source level
14	no	no	Program profiles Limited control
15	-	no	At source level
16	yes	no	Execution counts Allows some abbreviations for most commands Good trap control

Appendix B

Syntax of BIAS

Metalanguage

::= and | are used as in standard BNF
Non-terminals are in lower-case using - as a connector
(and) are used for grouping within the meta-language
OPT-notion ::= notion | empty
notion-*LIST* ::= notion | notion , notion-*LIST*
{notion} ::= notion | notion {notion}

Language

```
command              ::= immediate-command
                       | trap-command
                       | editing-command

immediate-command    ::= statement

trap-command         ::= break-command
                       | trace-command
                       | unbreak-command
                       | untrace-command
                       | option-command
                       | continue-command

editing-command      ::= insertion
                       | delete-command
                       | list-command
                       | make-command
                       | get-command
                       | save-command
                       | remove-command
                       | run-command
```

```
break-command       ::= ( BREAK | B )  OPT-trap-part  OPT-control-part
                        OPT-( : statement )

trace-command       ::= ( TRACE | T )  OPT-trap-part  OPT-control-part
                        OPT-( : statement )

unbreak-command     ::= ( UNBREAK | UNB )  OPT-trap-part

untrace-command     ::= ( UNTRACE | UNT )  OPT-trap-part

option-command      ::= ( SET | RESET )  option-LIST

continue-command    ::= ( CONTINUE | CONT | C )  OPT-line-num

insertion           ::= line-num  statement

delete-command      ::= ( DELETE | DEL )  line-num-seq-LIST

list-command        ::= ( LIST | L )  line-num-seq-LIST

make-command        ::= ( MAKE | M )  file-name

get-command         ::= ( GET | G )  file-name

save-command        ::= ( SAVE | S )  OPT-file-name

remove-command      ::= ( REMOVE | REM )  OPT-file-name

trap-part           ::= OPT-line-num-seq-LIST  OPT-statement-type-LIST
                      | ( identifier | variable )-LIST

control-part        ::= OPT-( ON condition )  OPT-( BY expression )
                        OPT-( FOR expression )

line-num-seq        ::= line-num  OPT-( - line-num )

option              ::= BREAK | B
                      | TRACE | T
                      | XCOUNT
```

The following definitions are language or operating-system dependent. The statement-types described are for STILL

file-name ::= any valid file title

statement ::= any valid statement in the host language (STILL)

line-num ::= any positive integer with 8 or less digits

statement-type ::= BEGIN
| END
| CASE
| ESAC
| LOOP
| OD
| ASSIGN
| READ
| WRITE
| CALL
| RETURN

variable ::= any valid variable in the host language that has been declared

identifier ::= any valid identifier in the host language that has been declared

expression ::= any valid expression yielding an arithmetic value

condition ::= any valid expression yielding a boolean value

Appendix C

Syntax of STILL

Metalanguage as for Appendix B

```
program                ::= statement

statement              ::= simple-statement
                         | compound-statement

simple-statement       ::= assignment-statement
                         | procedure-assignment
                         | return-statement
                         | read-statement
                         | write-statement
                         | invocation-statement
                         | empty

compound-statement     ::= block
                         | case-statement
                         | loop-statement
                         | proc-initialisation

assignment-statement   ::= variable := expression

procedure-assignment   ::= proc-id := proc-id

return-statement       ::= RETURN OPT-expression OPT-( FROM label )

read-statement         ::= READ OPT-SAME OPT-( variable | : format |
                                 variable : format )-LIST

write-statement        ::= WRITE OPT-SAME OPT-( expression | : format |
                                 expression : format )-LIST

invocation-statement   ::= proc-id OPT-( ( actual-parameter-LIST ) )
```

```
block                 ::= BEGIN OPT-label ← OPT-{ declaration ← }
                          { statement ← } END

case-statement        ::= ( CASE | IF ) ← { guard ← { statement ← } }
                          OPT-( ELSE ← { statement ← } )( ESAC | FI )

loop-statement        ::= OPT-( FOR identifier ) OPT-( FROM expression )
                          OPT-( BY expression ) OPT-( ( TO | DOWNTO )
                          expression ) OPT-( WHILE expression ) DO ←
                          OPT-{ statement ← } OD

proc-initialisation   ::= proc-id := OPT-( ( formal-parameter-LIST ) )
                          proc-type : ← statement

declaration           ::= type ( identifier OPT-( ( = | := )
                          expression ) )-LIST
                        | [ ( expression : expression )-LIST ] type
                          identifier-LIST
                        | PROC OPT-( ( formal-type-LIST ) ) proc-type
                          identifier-LIST

type                  ::= REAL
                        | INT
                        | BOOL

proc-type             ::= type
                        | VOID

formal-type           ::= type
                        | REF OPT-( [ empty-LIST ] ) type
                        | PROC OPT-( ( formal-proc-ptype-LIST ) )
                          proc-type

formal-proc-ptype     ::= type
                        | REF OPT-( [ empty-LIST ] ) type

formal-parameter      ::= formal-type identifier-LIST
```

```
actual-parameter      ::= identifier
                        | variable
                        | expression

guard                 ::= : expression

expression            ::= primary OPT-{ binary-operator primary }

primary               ::= OPT-{ unary-operator } operand

operand               ::= literal
                        | variable
                        | ( expression )
                        | proc-id OPT-( ( actual-parameter-LIST ) )

variable              ::= identifier OPT-( [ expression-LIST ] )

proc-id               ::= identifier

label                 ::= identifier

format                ::= ( A | H | I | L | R | T | X ) expression
                        | /

literal               ::= TRUE | FALSE
                        | string
                        | number

empty                 ::=
```

The non-terminals number and identifier are as for most languages, and are not detailed here.
The symbol ← denotes an end-of-line
The binary-operators and unary-operators are detailed in section 2.6

Appendix D

Pseudo-machine Instructions

1 Memory Access

STOD		store destructive: B register must be an IRW
STON		store non-destructive: leaves value in A register
NAMC	couple	name call: load absolute address given by couple
VALC	couple	value call: load value derived from couple
INDX	numsubs	index: load address described by descriptor and subscripts
LOAD		load value at IRW on top of stack
MPCW	address	make PCW: with return address
MDVW		make DVW: bounds at top of stack
MDSC		make DESC: and allocate storage

2 Procedures and Branches

MKST		mark stack: load MSCW
ENTR		enter procedure using PCW for return address
RETN	level	return: to the level indicated
BLCK	address	enter block using address for return address
ENDB		end block: will not require return address
JUMP	inc	jump: over the next inc bytes
JMPT	inc	jump over next inc bytes if top of stack is true
JMPF	inc	jump over next inc bytes if top of stack is false

3 Stack Manipulation

STAG	tag	set tag: of top of stack to tag
ZERO		load zero into top of stack (false)
ONE		load one into top of stack (true)
LIT8	value	load literal one byte
LT16	value	load literal two bytes
LT48	value	load literal one word
POP		decrement top of stack register

PUSH	increment top of stack register
DUPL	duplicate: push top of stack onto stack
XCHN	exchange: A and B registers

4 Binary Operators

For all these, B := B op A; T := T - 1

ADD, SUBT, MULT, DIVD, IDIV, MOD, POWR	(arithmetic)
LAND, LOR	(logical)
LEQ, LSS, GEQ, GTR, EQL, NEQ, IS, ISNT	(relational)

5 Unary Operators

For all these except RAND, A := op A;

NEG	
LNOT	logical not
SIN, COS, ATAN	
SIGN	
RAND	generate random number on top of stack
NTGR	integerize rounded
NTIA	integerize truncated
LOG	
SQRT	

6 I/O

CLIN	clear input buffer
CLOT	clear output nuffer
RDIN	read from file to input buffer
WROT	write from output buffer to file
RSIN	reset input buffer for rereading
WROS	write to the same record as last write
ITOS	input top of stack: from input buffer, formatted
OTOS	output top of stack: to output buffer, formatted
OSTR len,string	output string: given with length first

7 Trap Handling

TRAP	link threaded code, check for trap (section 3.2)
STRT	enter block using current address as return address
FNSH	exit procedure, returning to supervisor
END	return to the supervisor

Bibliography

Ashb 73 Ashby G. et al.
'Design of an interactive debugger for FORTRAN: MANTIS'
Software Practice and Experience 3 p65-74 1973

Balz 69 Balzer R. M.
'EXDAMS - EXtendable Debugging And Monitoring System'
AFIPS Proc. SJCC Vol 34 p 567-580 1969

Barr 75 Barrer P. J.
'The Use of Alphanumeric Display Terminals'
Thesis Massey University 1975

Barr 69 Barron D. W.
'A Note on Program Debugging in an On-line Environment'
Computer Journal 12 p 104 1969

Bate 74 Bate D. G.
'Design and Implementation of an Interactive Test-bed'
Software Practice and Experience 4 p 91-109 1974

Bern 68 Bernstein W. A. and J. T. Owens
'Debugging in a time-sharing environment'
AFIPS Proc. FJCC Vol 33 p 7-14 1968

Blai 71 Blair J.
'Extendable Non-Interactive Debugging' p 93-115 in
Rustin R. ed. Debugging Techniques in Large Systems
Prentice-Hall Englewood Cliffs N.J. 1971

Brow 73 Brown A. R. and W. A. Sampson
'Program Debugging' Macdonald London 1973

Bull 72 Bull G. M.
'Dynamic Debugging in BASIC'
Computer Journal 15 p 21-24 1972

Buxt 69 Buxton J. N. and F. L. Bauer (ed)
'Software Engineering Techniques' NATO Science Committee
Birmingham 1969

Carp 76 Carpenter B. I. and L. K. Thomas
'User Attitudes to Remote Interactive Terminals'
Massey University Computer Unit Report 28 1976

Davi 75 Davis A. M.
'An Interactive Analysis System For Execution-Time Errors'
Thesis Ph.D. University of Illinois at Urbana-Champaign 1975

Dijk 69 Dijkstra E. W.
'Structured Programming' in Buxton J. N. and F. L. Bauer (ed)
Software Engineering Techniques NATO Science Committee
Birmingham 1969

Dora 72 Doran R. W. and G. Tate
'Approach to Structured Programming'
Massey University Computer Unit Report 10 1972

Dora 73 Doran R. W.
'Introduction to Stack Machines'
Massey University Computer Unit Report 11 1973

Elsp 72 Elspas B. et.al.
'An Assessment of Techniques for Proving Program Correctness'
Computing Surveys 4 p 97-149 1972

Evan 66 Evans T. G. and D. L. Darley
On-line Deugging Techniques: A Survey'
AFIPS Proc. FJCC Vol 29 p 37-50 1966

Ferg 63 Ferguson H. E. and E. Berner
'Debugging Systems at the Source Language Level'
Communications of the ACM 6 p 430-432 1963

Gain 69 Gaines R. S.
'The Debugging of Computer Programs'
Thesis Ph.D. Princeton University 1969

Goul 75 Gould J. D.
'Some Psychological Evidence on How People Debug Computer Programs'
International Journal of Man-Machine Studies 7 p 151-182 1975

Gris 70 Grishman R.
'The Debugging System AIDS'
AFIPS Proc. SJCC Vol 36 p 59-64 1970

Gris 71 Grishman R.
'Criteria for a Debugging Language' p 57-75 in Rustin R. (ed)
Debugging Techniques in Large Systems Prentice-Hall
Englewood Cliffs N.J. 1971

Gris 73 Griswold R. E.
'A SNOBOL 4 Primer' Prentice-Hall Englewood Cliffs N.J.
1973

Grov 74 Groves L. J.
'The Provision of Deugging Facilities for High Level Languages'
Thesis M.Sc. Massey University 1974

Hans 71 Hansen W. J.
'User Engineering Principles for Interactive Systems'
AFIPS Proc. FJCC Vol 39 p523 1971

King 71 King J.
'A Verifying Compiler' p 17-40 in Rustin R. (ed)
Debugging Techniques in Large Systems Prentice-Hall
Englewood Cliffs N.J. 1971

Kuls 71 Kulsrud H. E.
'Extending the Interactive Debugging System HELPER'
p 77-91 in Rustin R. (ed) Debugging Techniques in Large Systems prentice-Hall Englewood Cliffs N.J. 1971

Lask 74 Lasker D. M.
'An Investigation of a New Method of Constructing Software'
University of Toronto Technical Report CSRG-38 1974

Lond 70 London R. L.
'Bibliography on Proving the Correctness of Programs'
p 569-580 in Machine Intelligence 5 Oliver & Boyd
Edinburgh 1970

Oste 76 Osterweil L. J. and L. D. Fosdick
'DAVE - A Validation Error Detection and Documentation
System for FORTRAN Programs'
Software Practice and Experience 6 p 473-486 1976

Pier 74 Pierce R. H.
'Source Language Debugging on a Small Computer'
Computer Journal 17 p 313-317 1974

Pool 73 Poole P. C.
'Debugging and Testing' p 278-318 in Goos G. and
J. Hartmanis (ed) Software Engineering - An Advanced Course
Springer-Verlay Berlin 1973

Rain 73 Rain M.
'Two Unusual Methods For Debugging System Software'
Software Practice and Experience 3 p 61-63 1973

Rand 75 Randell B.
'System Structure for Software Fault Tolerance'
University of Newcastle-Upon-Tyne Computing Laboratory
Technical Report Series No. 75 1975

Rish 70 Rishel W. J.
'Incremental Compilers'
Datamation 16 (1) p 129-136 1970

Ryan 66 Ryan J. L. et. al.
'A Conversational System for Incremental Compilation in a
Time-Sharing Environment'
AFIPS Proc. FJCC Vol 29 p 1-21 1966

Sack 68 Sackman H.
'Time-Sharing vs Batch Processing: the Experimental Evidence'
AFIPS Proc. SJCC Vol 32 p 1-10 1968

Satt 72 Satterthwaite E.
'Debugging Tools for High Level Languages'
Software Practice and Experience 2 p 197-217 1972

Schw 71 Schwartz J. T.
'An Overview of Bugs' p 1-16 in Rustin R. (ed)
Debugging Techniques in Large Systems Prentice-Hall
Englewood Cliffs N.J. 1971

VanT 74 Van Tassel D.
'Program Style, Design, Efficiency, Debugging and Testing'
Prentice-Hall Ebglewood Cliffs N.J. 1974

Wirt 71 Wirth N.
'Program Development by Stepwise Refinement'
Communications of the ACM 14 p 221-227 1971

Wolm 72 Wolman B. L.
'Debugging PL/I Programs in the Multics Environment'
AFIPS Proc. FJCC Vol 41 p 507-514 1972

Your 75 Yourden E.
'Techniques of Program Structure and Style'
Prentice-Hall Englewood Cliffs N.J. 1975

Zelk 73 Zelkowitz M. V.
'Reversible Execution'
Communications of the ACM 16 p 566 1973

Manuals

Burr 74 A Burroughs B6700/B7700 Algol Language Reference Manual
Form no. 5000649

Burr 74 F Burroughs B6700/B7700 FORTRAN Reference Manual
Form no. 5000458

Burr 76 Burroughs B6700 P notes/D notes merged documentation
Form no. 5001241