SCIENCE RESEARCH COUNCIL

RUTHERFORD LABORATORY
ATLAS COMPUTING DIVISION

FR80 TECHNICAL PAPER 21

FR80 DRIVER Software Construction

issued by
R W Witty

10 December 1975

DISTRIBUTION:

F R A Hopgood
R Brandwood
I Buchanan
A W Burraston
M F Chiu
P A Dewar
L O Ford
A H Francis
J R Gallop
P M Nelson
D Ralphs
D C Sutcliffe
R E Thomas
R W Witty

1. INTRODUCTION

New software, FR80 DRIVER, is under construction to ease the current maintenance problems with the existing III Displayer, and to provide new facilities (FR80 DRIVER is outlined in FR80 Discussion Paper 15).

This paper describes the methods of design and construction in use with DRIVER. These methods include designing by Step Wise Refinement and its representation by flowcharts, the encoding of these flowcharts into an intermediate language, and the translation of this language into an FR80 program. The paper closes with a discussion of DRIVER programming standards and style.

2. FLOWCHARTING AND STEP WISE REFINEMENT

2.1 Conventional Flowcharting

FR80 SYSLOG was designed and implemented by top down, disciplined programming techniques. Flowcharts of the program were produced at successive levels of detail using the method of Step Wise Refinement [7]. The final flowchart was sufficiently detailed for FR80 Assembler code to be produced almost mechanically. During the design of SYSLOG several inadequacies of conventional flowcharting were encountered:-

(1) they are difficult to draw and lay out as they spread out in all directions from a given starting point. This makes formatting on a page a problem, often requiring redrawing.

(2) they have complex symbols which are hard to draw neatly freehand.

(3) if a template is used then fitting text into the boxes is a grave problem.

(4) they cannot be drawn 'naturally'; programming language source is easier because it follows the normal lexical 'directions' of English.

(5) they are difficult to convert to a machine readable form.

(6) they are difficult to draw 'automatically'.

(7) they are usually drawn on A4 paper which is too small to convey the logic of a complex module. Arbitrary, redundant branches are caused by page boundaries making a disjointed flowchart that is hard to visualise overall.

(8) they fail to represent such properties as parallelism because sequential actions are spread out over both dimensions of the page.

(9) they positively encourage undisciplined branches and 'cluttered' logic because it is so easy to draw branches in all directions.

(10) they fail completely to show how the final detailed design has been achieved through Step Wise Refinement (SWR).

The above problems were overcome by developing a variant of flowcharting called Dimensional Flowcharting, described below. The final design of SYSLOG was actually represented by this method.

2.2 Dimensional Flowcharting

Consider the CASE statement

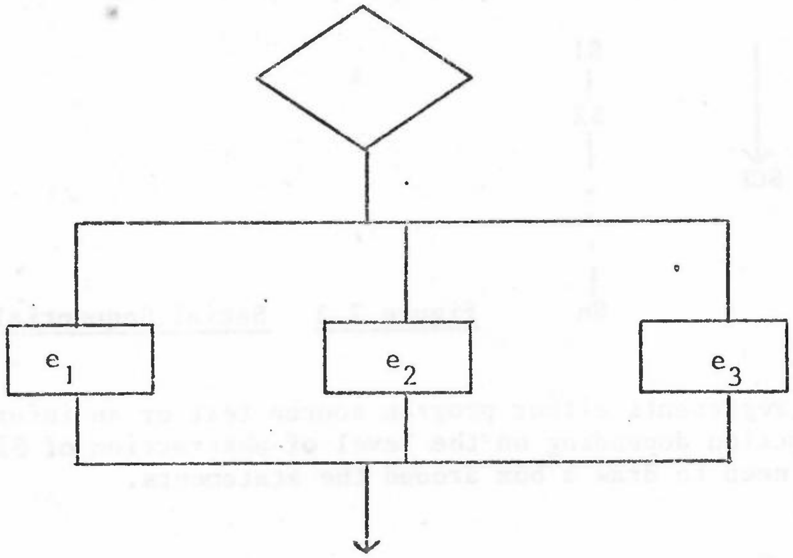


figure 2.1 Conventional CASE statement

This is a well-disciplined construct having one entry point and one exit point. It shows how the conventional direction of sequential control is from the top to the bottom of a flowchart. It also reveals that the executable code blocks e_1, e_2, e_3 exit as parallel alternatives, although only one is executed to the exclusion of the others. This is the selective CASE statement, the most common form. In a generalised CASE statement whose semantics are that each and every true case is executed, then the parallelism is more obvious [3]. (Note that IF-THEN-ELSE is a simple CASE statement, the cases being 'boolean expression is true' and 'boolean expression is false'.)

From this example we can postulate that the 'dimensions' of a flowchart are sequential control flow (SCF) and parallel execution (P), figure 2.2.

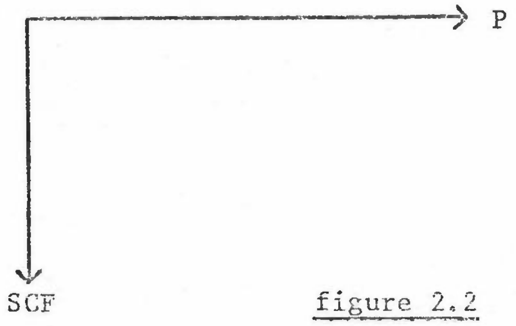


figure 2.2 2-D axes

2.2.1 Sequential Statements

For the flow of control to be always top to bottom we must impose a discipline on branch instructions. Every program must be a sequence of statements, having only one entry and one exit point, such as IF-THEN-ELSE, CASE and LOOP. Hence in Dimensional Flowcharting a sequence of statements S1,S2,.....,Sn is shown as

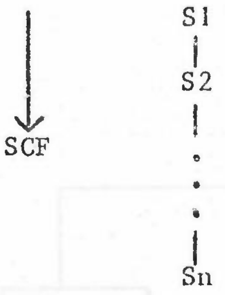


figure 2.3 Serial Sequential Statements

Any Si represents either program source text or an informal description of an action depending on the level of abstraction of Si. There is now no need to draw a box around the statements.

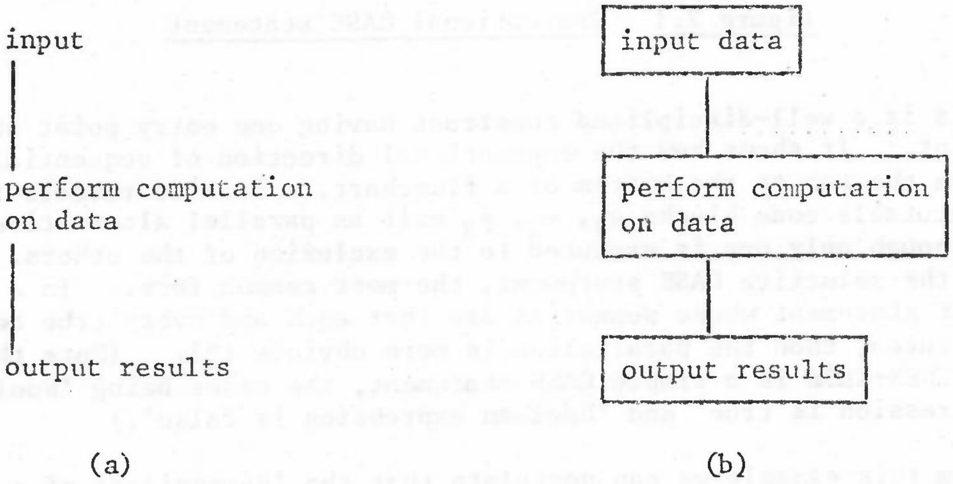


figure 2.4 Contrasting Representations of Sequential Statements

2.2.2 CASE

The CASE statement is drawn as

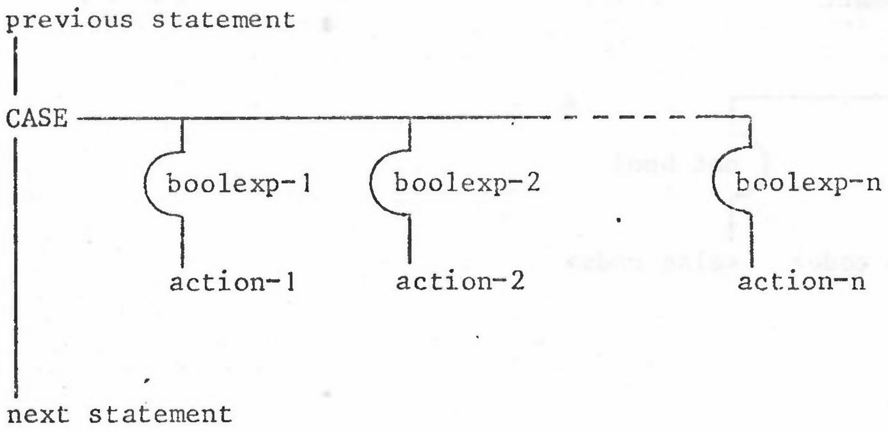


figure 2.5 Dimensional CASE Statement

The selection mechanism is controlled by the Conditional statement

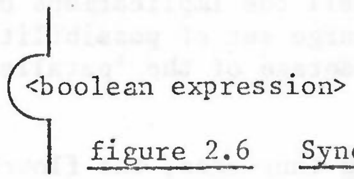


figure 2.6 Synchronisation Mechanism

Its semantics are that if <boolean expression> is true then control passes on down the vertical line (crosses the bridge [12]). If <boolean expression> is false then control is prevented from continuing along the vertical branch. This is a general mechanism which is used in other constructs.

Returning specifically to the CASE statement, its action is that if one or more vertical branches have true <boolexp-i>s then the corresponding <action-i>s are executed. Control only passes to the statement after the CASE when all of these have finished execution. This shows the parallel dimension of the flowchart to its full advantage as control can be imagined to instantaneously flow along a horizontal line and down into each vertical line as though an infinite number of parallel processors were available to execute the statement. If all vertical branches are blocked by false <boolexp-i>s then the whole statement is terminated and control passes to the next sequential statement.

The simple boolean blocking statement described above is good enough to represent purely serial algorithms. If a designer were to use this flowcharting method for constructing real parallel algorithms it is likely that he would add his own synchronisation devices. The main point being made in this section is that the rigorous use of the two dimensional page elegantly represents parallel code and sequential ordering.

2.2.3 IF-THEN-ELSE

Using the above scheme, IF-THEN-ELSE is

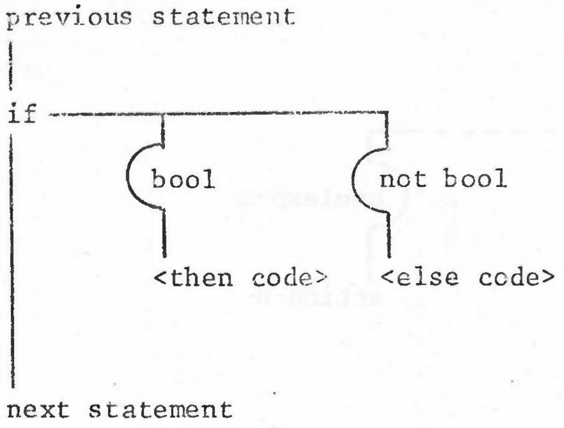


figure 2.7 Dimensional IF-THEN-ELSE

Note that now, because of the explicit parallelism, the <then code> and <else code> blocks must both be conditionally executed. This forces the programmer to consider fully all the implications of the <else code> operating on the (usually) large set of possibilities <not bool>, and is put forward as an advantage of the 'parallel' way of thinking.

Obviously for normal, serial programming languages, the flowchart will be encoded using the conventional 'jump to <else code> if boolean is false' technique.

2.2.4 LOOP

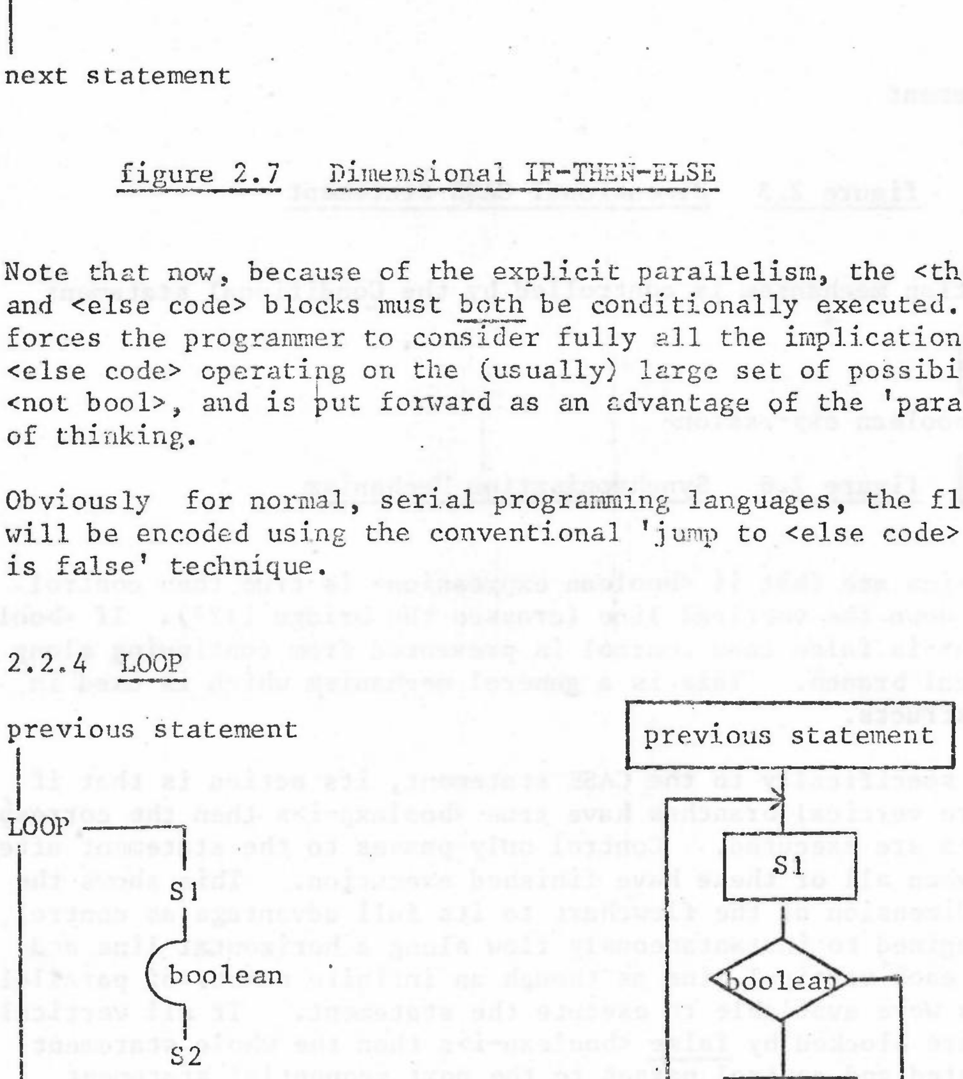


figure 2.8 Contrasting Loops

In the LOOP statement, figure 2.8a, the loop body (S1, conditional exit, S2) is repeated while <boolean> is true. Loops are discussed further in section 3.4. To help differentiate between a sequence of statements which is repeated (loop body) and one which is not (block body, section 3.5) it is sometimes helpful to use the symbols '*' and '⏚' which indicate repetition and termination respectively. However they are usually omitted. Figure 2.9 demonstrates their use.

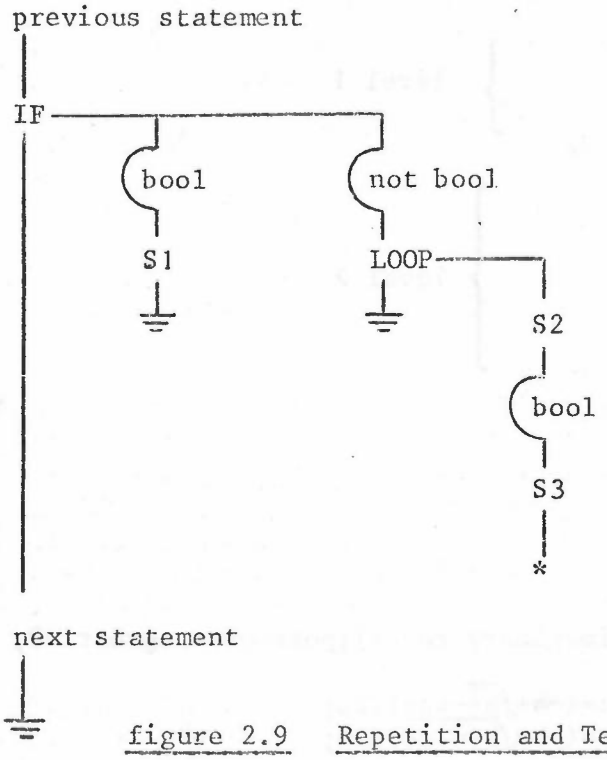


figure 2.9 Repetition and Termination Symbols

2.2.5 Branch

Dimensional Flowcharting can encompass direct branches but it clearly shows how they spoil the elegance and discipline of the design, and how their occurrence, in practice, seems unnatural and unnecessary. This feeling grows after using the disciplined Dimensional Flowcharting method; one quickly forgets that GOTOs ever existed. The FR80 SYSLOG design was free from direct GOTOs, and it was greatly improved by thinking more deeply about the reliability of the control flow.

If an explicit GOTO must be used, it can be shown as figure 2.10.

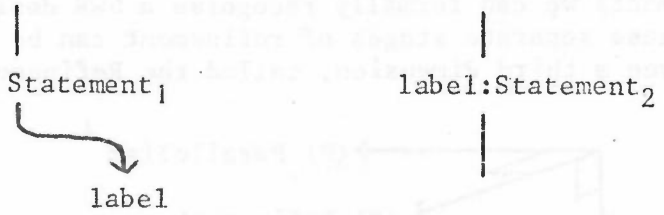


figure 2.10 Branch

The resemblance between the GOTO notation and the Devil's tail is not coincidental.

2.3 Step Wise Refinement

A property of Step Wise Refinement (SWR) is that the most up-to-date design is expressed in terms of the lowest level reached so far. This means that, at worst, the derivation of large parts of the design are 'lost' as they are refined, or at best, can only be deduced from a study of separate flowcharts of the various intermediate stages (figure 2.11). Is there a unified method of representation which will preserve what amounts to a record of the designer's thoughts?

```
solve quadratic equn.                                  }  level 1


input data                                             }
|                                                      }
perform computation on data                            }  level 2
|                                                      }
output results                                         }


read A;                                                            }
                                                                   }
read B;                                                            }
                                                                   }
read C;                                                            }
                                                                   }
if descriminant <0 then print (imaginary roots);posroot:=negroot:=0;
                                                                   }  level
                  else posroot:=$(-b+\sqrt{b^2-4ac})/2a$;          }
                       negroot:=$(-b-\sqrt{b^2-4ac})/2a$;          }
                                                                   }
print A;                                                           }
      B;                                                           }
      C;                                                           }
      posroot;                                                     }
      negroot;                                                     }
stop;                                                              }
```

figure 2.11 SWR Example

By using the Algol-like rule that every program is a single (compount) statement which can be recursively split into sequences of statements (the process of refinement) we can formally recognise a SWR design (see section 2.5). These separate stages of refinement can be connected if the flowchart is given a third dimension, called the Refinement(R).

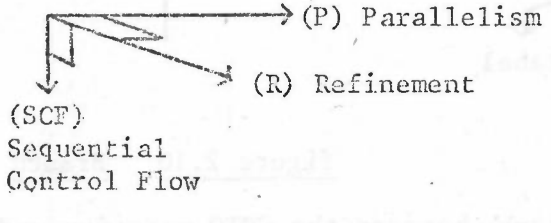


figure 2.12 3-D Axes

The example given in figure 2.11 now becomes figure 2.13.

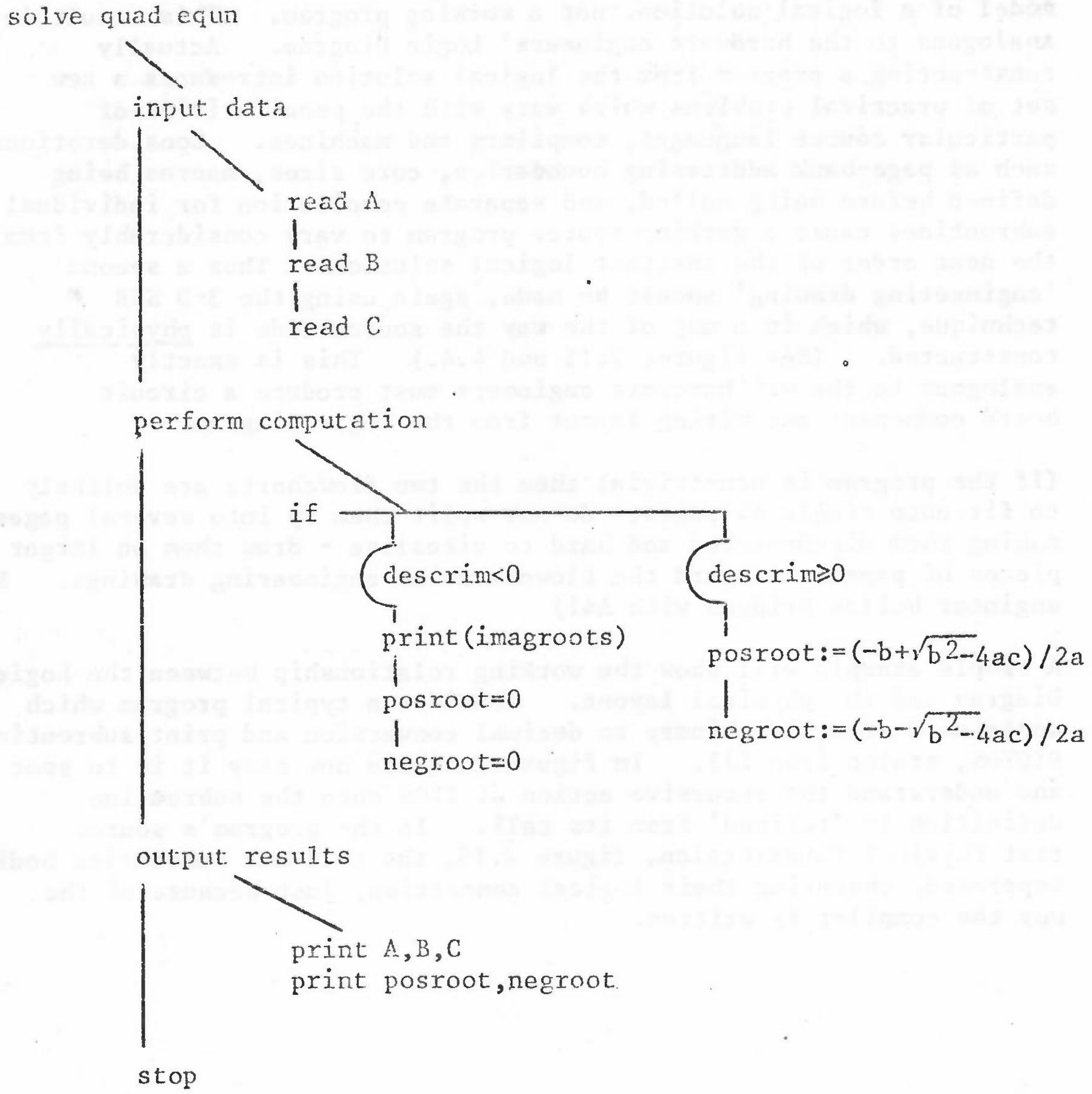


figure 2.13 3-D Flowchart

Now one can study the program's design at any desired level of abstraction; one may study the design at varying stages of refinement as the area of interest changes, digging deeper down into the details of, say, 'input data' to see exactly what is happening. Having once understood the action of the input phase one need never again go deeper than the 'input data' level to recall the program's action at this point.

If one regards the higher levels of abstraction as comments about the lowest level, then this hierarchy of comments may be 'folded' or 'linearised' into the source code text, whence the correspondence between the flowchart and the source code is exact in the refinement sense and the source program-flowchart couple becomes much easier to read and understand. This proposal was followed with FR80 SYSLOG and works well in practice (see example DRIL program, section 4.2).

2.4 Design and Construction

Tackling a problem by creating a 3-D flowchart via SWR produces a model of a logical solution, not a working program. This result is analogous to the hardware engineers' Logic Diagram. Actually constructing a program from the logical solution introduces a new set of practical problems which vary with the peculiarities of particular source languages, compilers and machines. Considerations such as page-bank addressing boundaries, core sizes, macros being defined before being called, and separate compilation for individual subroutines cause a working source program to vary considerably from the neat order of the abstract logical solution. Thus a second 'engineering drawing' should be made, again using the 3-D SWR technique, which is a map of the way the source code is physically constructed. (See figures 2.15 and 4.4.) This is exactly analogous to the way hardware engineers must produce a circuit board component and wiring layout from the Logic Diagram.

(If the program is non-trivial then the two flowcharts are unlikely to fit onto single A4 pages; do not split them up into several pages, making them disconnected and hard to visualise - draw them on larger pieces of paper. Regard the flowcharts as engineering drawings. No engineer builds bridges with A4!)

A simple example will show the working relationship between the Logic Diagram and the physical layout. Consider a typical program which contains a recursive binary to decimal conversion and print subroutine, BIOTDS, stolen from [1]. In figure 2.14 see how easy it is to spot and understand the recursive action of ITOS when the subroutine definition is 'refined' from its call. In the program's source text Physical Construction, figure 2.15, the call and subroutine bodies are separated, obscuring their logical connection, just because of the way the compiler is written.

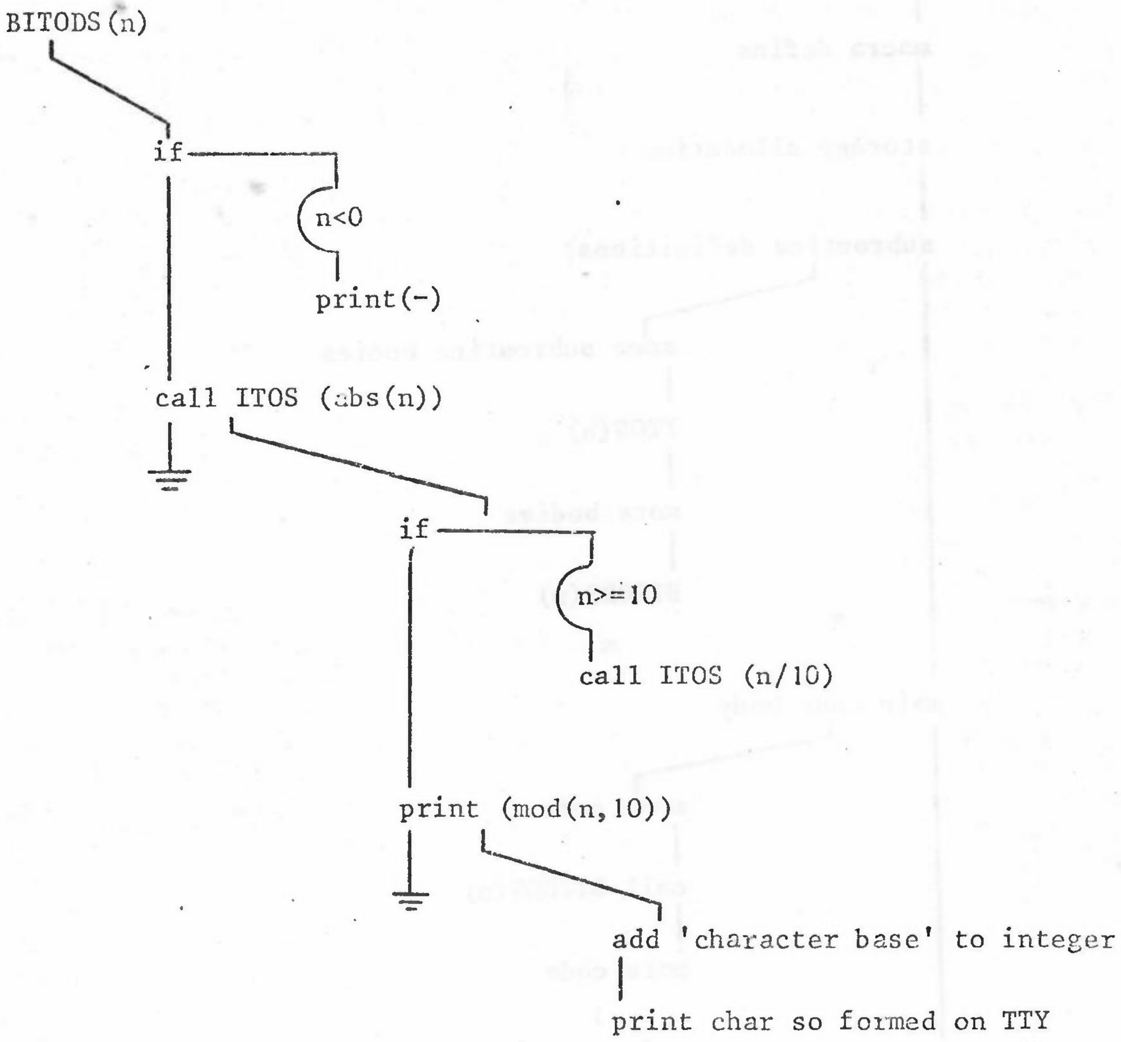


figure 2.14 Recursive Binary Integer to Decimal String Conversion

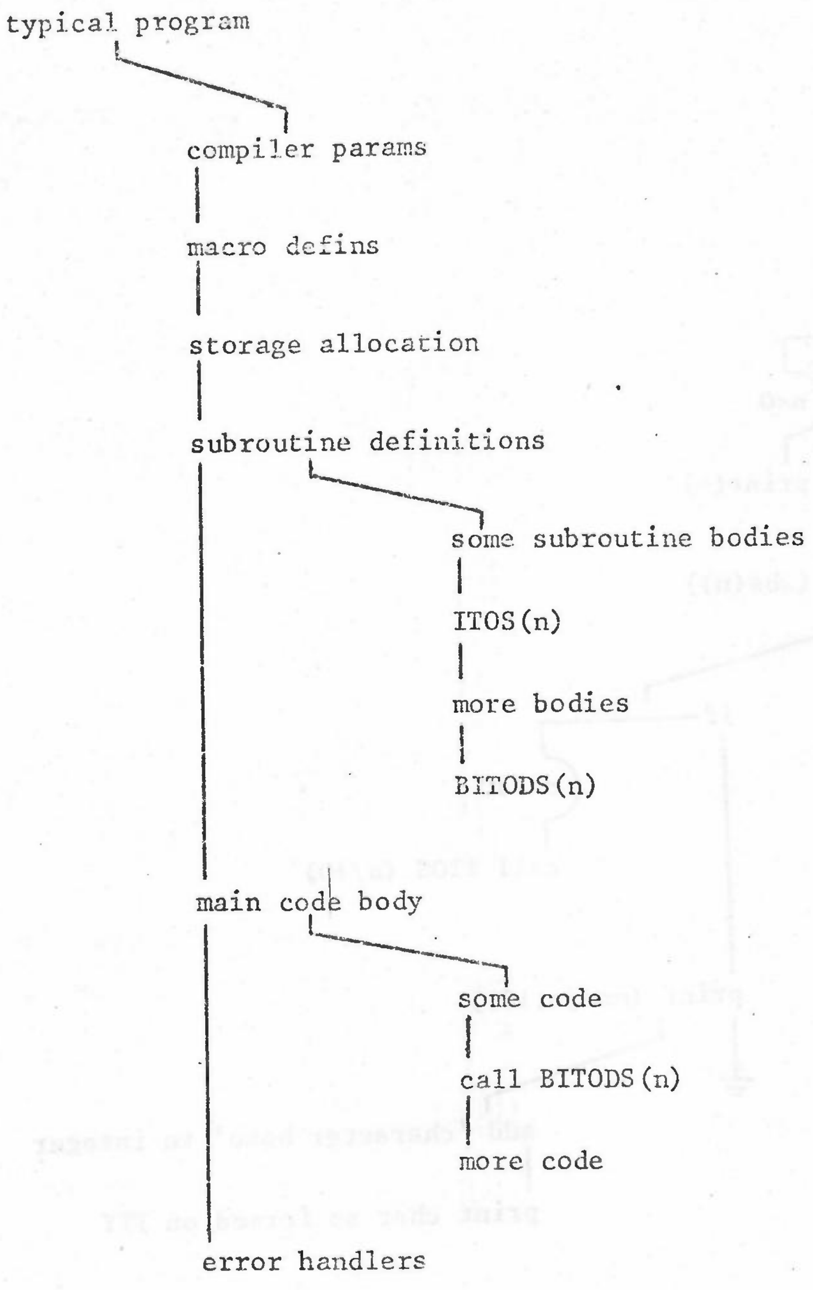


figure 2.15 Physical Construction of Typical Program

Using the Logic Design flowchart in conjunction with the Physical Construction flowchart and the source text with its 'folded' SWR comments greatly simplifies the problem of relating the physical program code to its logical design and action. Again, this observation is based on the experience gained from FR80 SYSLOG.

2.5 Flowchart Syntax

A direct benefit of introducing discipline into the design methodology is that a context free grammar can be defined which will generate 3-D flowcharts. Figure 2.16 is the TREE-META [9] definition of such a grammar. This property means a flowchart can be shown to conform to the design methodology and to the project standards which should be built into the grammar.

```
.META FLOW
FLOW = '.NAME' .ID ';' NODE '.END' .ID ';' ;
NODE = SERAL $( SERAL ) '#' ;
SERAL = '!' STMT (NEXTLEV / .EMPTY) $( PARAL ) ;
PARAL = '-' NODE ;
NEXTLEV = 'L' NODE ;
STMT = (.ACTST / EXSIT ) ;
ACTST = '=' .SR ;
EXSIT = '?' .SR ;
.END
```

figure 2.16 3-D Flowchart Syntax

2.6 Automated Drafting

It can be shown from the grammar (figure 2.16) that a flowchart is a tree. A simple tree-walk will produce a linear machine-readable version if the various straight lines and flowcharting symbols are encoded as unique identifiers. Such a tree-walk is similar to unparsing [2]. This human process is quick and easy, and the output matches the original drawing (figure 2.17).

The linear tree-walk can be fed into a syntax analyser (figure 2.16) for validation, and a straightforward recursive graphical algorithm can be constructed to produce high quality output on a plotting device by exploiting the grammar rules to draw the flowchart left to right, top to bottom, which makes the formatting very easy (it is 'context free'). The ease with which drafting may be automated is another major advantage of Dimensional Flowcharting.

In figure 2.17

! represents a vertical line, the SCF direction

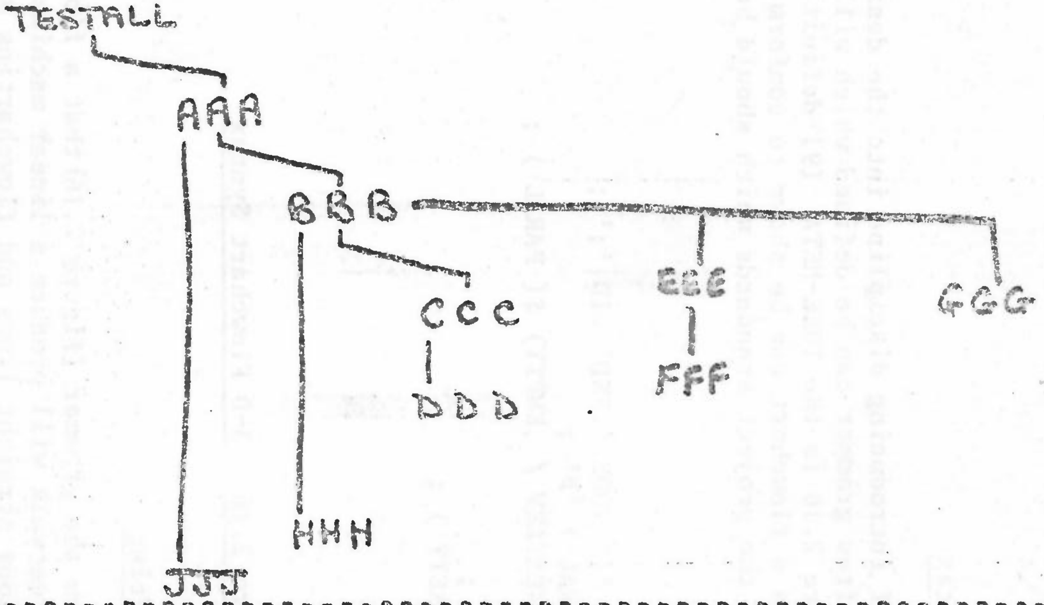
- represents a horizontal line, the P direction

L represents an angled line, the R direction

# represents the '⏚' symbol indicating the end of a series of sequential statements

figure 2.17 Machine Readable Flowchart

```
:GSIN21.FLOWA

#LISTING OF :GSIN21.TESTALL(1/) PRODUCED ON  7OCT75 AT 21.54.30

#OUTPUT ON SRC 6A G832V6   UNIT U15 BY JOB ':GSIN21.FLOWA' ON 14OCT75 AT 20.07.36

DOCUMENT    TESTALL

     0 .NAME TESTALL;
     1 | = 'AAA'
     2 L
     3 | = 'BBB'
     4 L
     5 | = 'CCC'
     6 | = 'DDD' #
     7 = | = 'EEE' | = 'FFF' #
     8 = | = 'GGG' #
     9 | = 'HHH' #
    10 | = 'JJJ' #
    11 .END TESTALL;
    12
    13 ****
    14
```



2.7 Summary of Advantages of Dimensional Flowcharting

1. Quick and easy to draw, by hand or machine, because drawn left to right, top to bottom, i.e. naturally.

2. Grammar ensures flowcharts are well-formed.

3. Grammar provides a mechanism for enforcing project standards.

4. Easy to convert to machine readable form via simple 'tree-walk' method.

5. Grammar makes automatic drafting very easy.

6. Statements may be any length.

7. Statements are not constrained to fit into boxes.

8. No need for templates.

9. Few special symbols.

10. Adaptable to any source code and machine.

11. Easy to introduce new features such as control constructs and synchronisation devices.

12. Shows inherent parallelism which is often not obvious from the source code.

13. Recursion easier to represent and understand.

14. 'Automatically' ensures will disciplined program design. Cluttered logic is prevented.

15. Encourages and facilitates SWR design.

16. Makes it easy for a 3rd party to understand how the design arrived at because SWR explicit.

17. One flowchart shows all 3 dimensions at once, making it possible (and easy) to visualise the whole program at varying levels of abstraction.

18. When both the logical design and the physical construction diagrams are drawn as large size 'engineering drawings' and when all the levels of abstraction are 'folded' into the source code as comments then the 3 documents work in unison to greatly improve the speed and understanding of the program by a 3rd party (and the designer himself).

3. DRIL

3.1 Introduction

FR80 SYSLOG was designed using high level control constructs. The FR80 only has an assembler so these control constructs had to be hand-compiled - a tedious, error-prone process. As DRIVER will be a much larger program than SYSLOG, it was decided to create a simple implementation language (DRIL), based on these control constructs, which could be easily translated into FR80 Assembler source code. DRIL is an aid to assembler programming, similar to but more powerful than macros, rather than a full scale high level language. A DRIL program should therefore be the easiest, fastest way to produce a correct FR80 Assembler program.

The aims of DRIL are

(1) a reduction in coding errors.
Cases, Loops and Boolean Expressions are difficult to hand-compile accurately first time, every time.

(2) an improvement in the 'readability' of the code.
Assembler versions of control constructs, especially their boolean expressions, are not easy to understand. DRIL's free format helps to highlight loop bodies.

(3) an improvement in the consistency of construction.
The tendency to locally optimise assembler code in SYSLOG caused a lack of uniformity which made 3rd party understanding more difficult.

(4) that the DRIL statements should reflect the 3-D flowcharting system (section 2) making SWR and reliable control flow easier to achieve so that DRIL, the Logic Design and Physical Construction flowcharts all work in harmony.

The object program produced by the DRIL translator (section 4) is an FR80 Assembler source code program because

(1) this simplifies the process of translation; the work of the assembler is not duplicated by the DRIL translator.

(2) the DRIVER program will be transportable to other sites in source form [8]. This is one reason why the DRIL translator attempts to produce an object program which looks like a hand written assembler program.

(3) this simplifies checking the translator.

(4) this allows incorporation of non-DRIL produced code at the assembler <u>source</u> level.

(5) macro-15 programs can also be produced from DRIL source programs.

A translator writing system is used to produce the DRIL translator because

(1) it is the fastest method of implementation; the effort is put into defining DRIL, not into producing a syntax analyser etc.

(2) it is the easiest method as it builds on existing work.

(3) it is the most reliable way as an already proven system is being used (Tree-Meta on the ICL 1906A [9]).

(4) it is the most flexible; DRIL and DRIVER are likely to evolve together and Tree-Meta allows DRIL to be modified easily.

(5) it allows software development to exploit the power of the ICL 1906A, freeing the FR80 for production work.

(6) Tree-Meta's flexible code generation system makes it possible to produce translator-commented code, neatly laid out with the source code comments retained, so that a DRIL object program is an FR80 Assembler source code program indistinguishable from a well written hand-coded equivalent.

An example showing the full range of DRIL source statements, their flowcharts and the code they produce is given in section 3.2. An actual DRIL module producing valid FR80 code is shown in section 4.2. The individual DRIL statements are explained below.

3.2 DRIL Statements

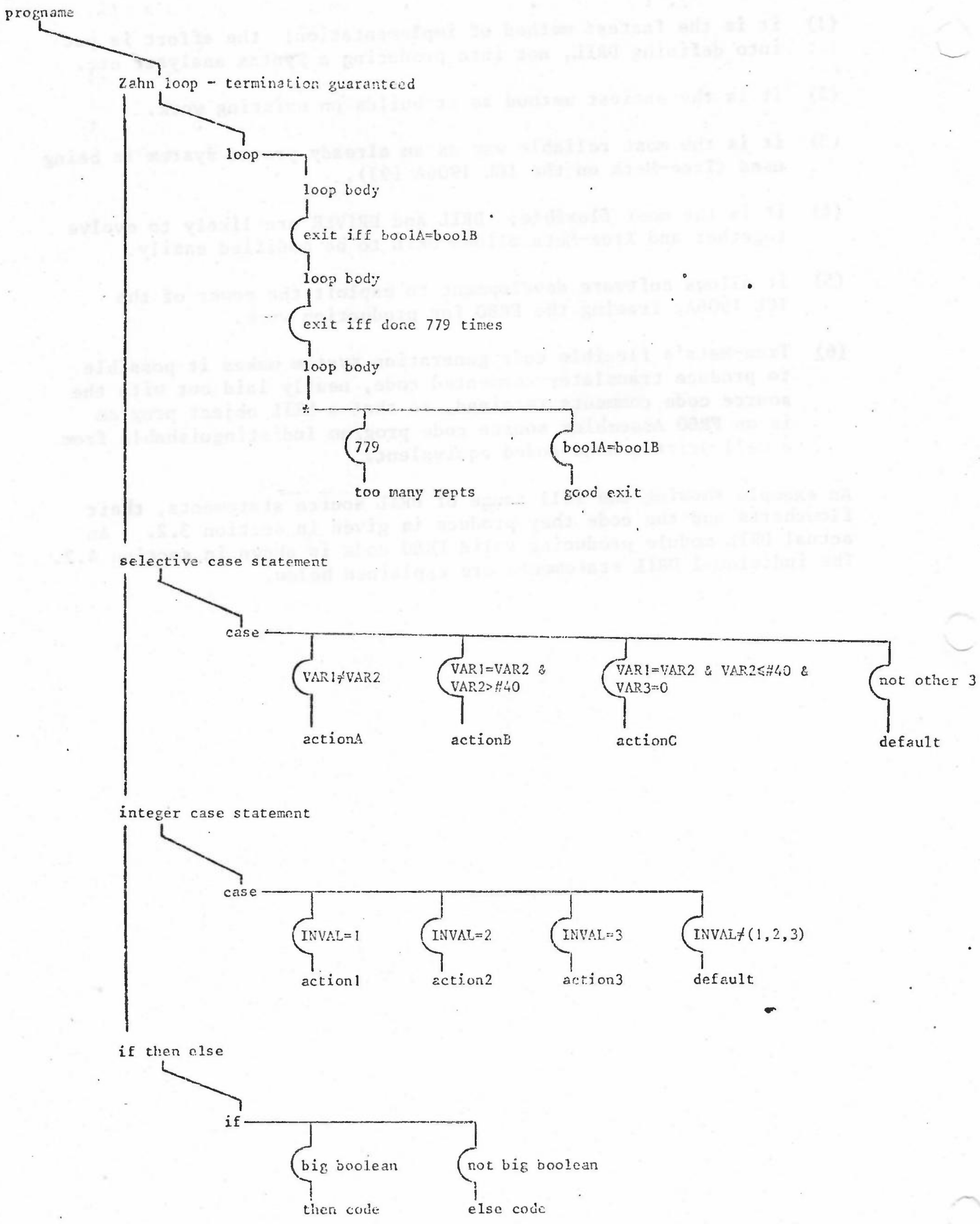
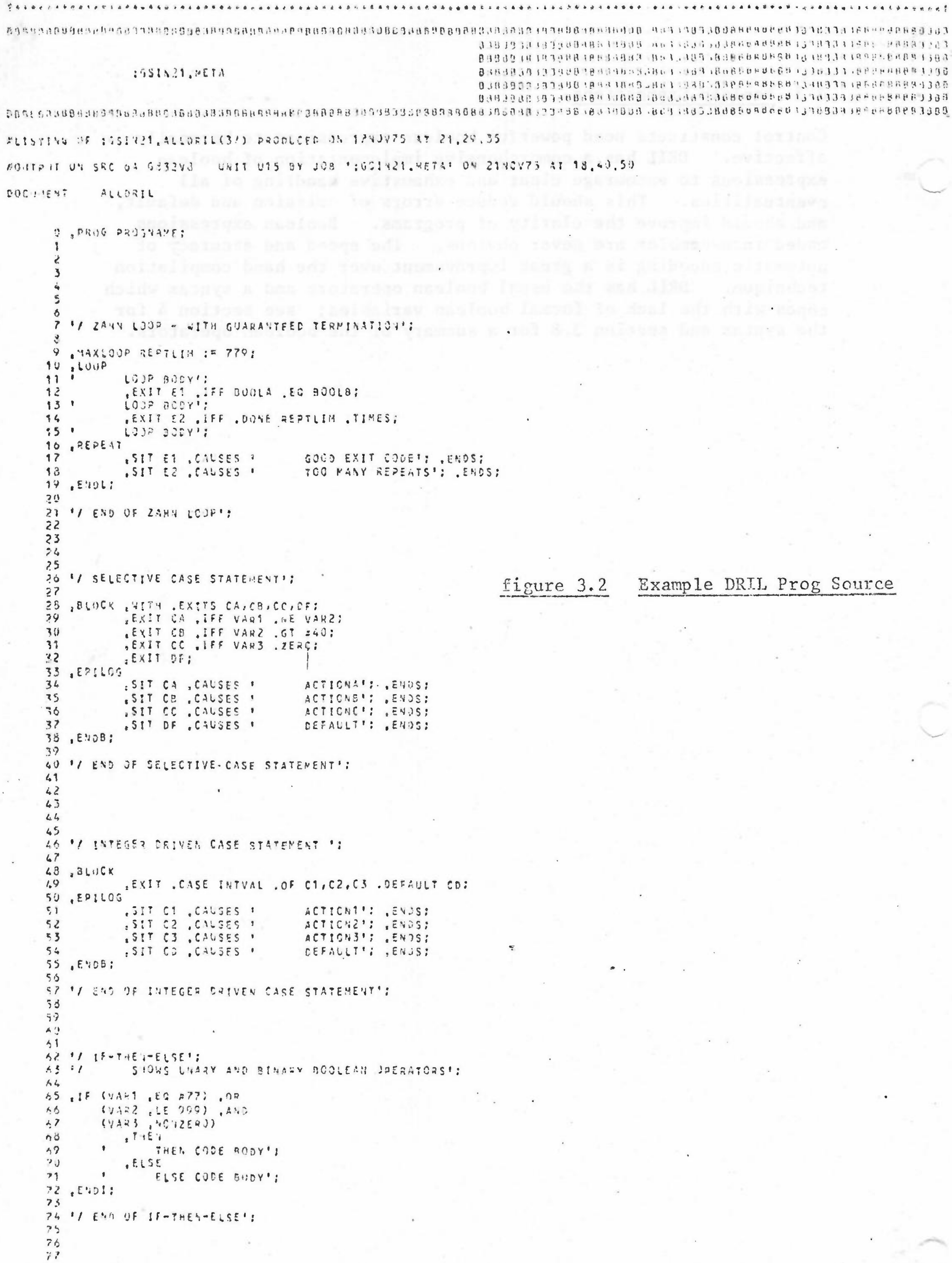


figure 3.1 3-D Flowchart of all DRIL Statements

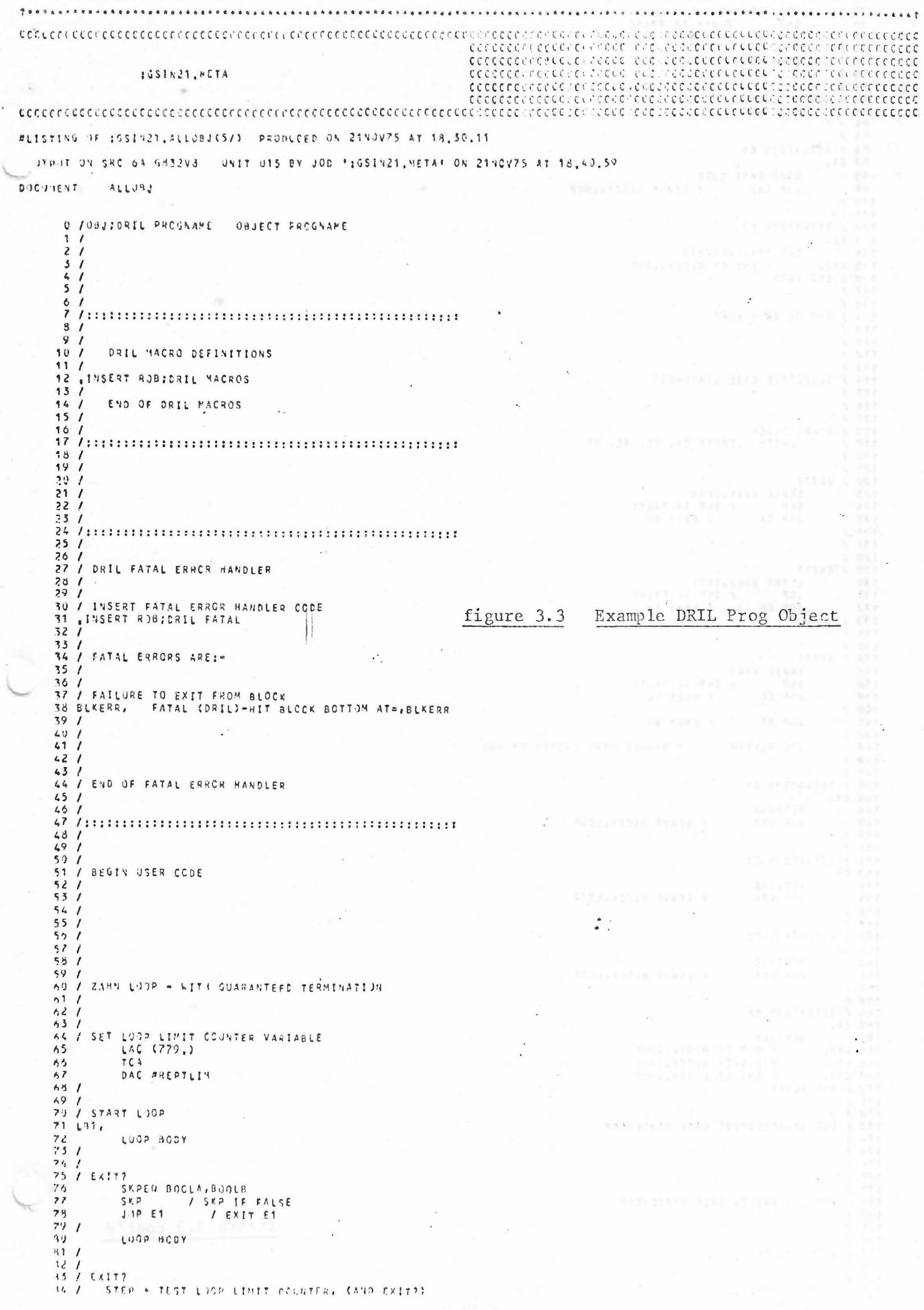
3.3 Boolean Expressions

Control constructs need powerful boolean expressions to be really effective. DRIL has a comprehensive implementation of boolean expressions to encourage clear and exhaustive handling of all eventualities. This should reduce errors of omission and default, and should improve the clarity of programs. Boolean expressions coded in assembler are never obvious. The speed and accuracy of automatic encoding is a great improvement over the hand compilation technique. DRIL has the usual boolean operators and a syntax which copes with the lack of formal boolean variables; see section 4 for the syntax and section 3.8 for a summary of the boolean operators.



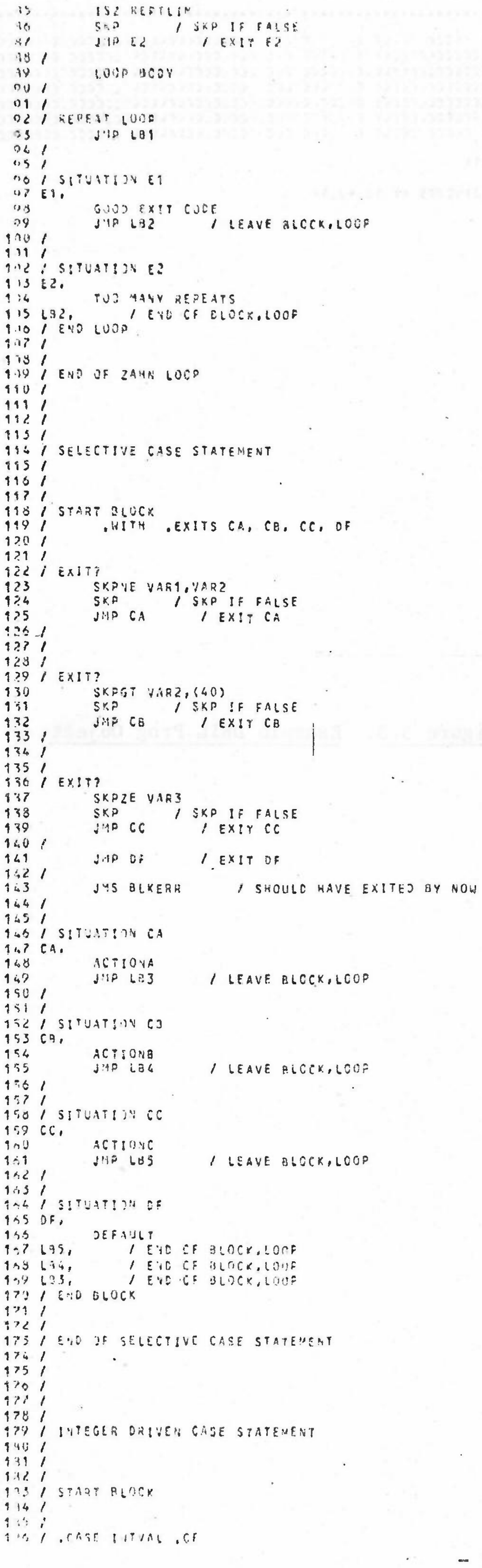
```
                    :GSIN21.META

#LISTING OF :GSIN21.ALLDRIL(3/) PRODUCED ON 17NOV75 AT 21.29.35

#OUTPUT ON SRC ON G332V3   UNIT U15 BY JOB ':GSIN21.META' ON 21NOV75 AT 18.40.59

DOCUMENT    ALLDRIL

 .PROG PROGNAME;






 '/ ZAHN LOOP - WITH GUARANTEED TERMINATION';

 .MAXLOOP REPTLIM := 779;
 .LOOP
 '        LOOP BODY';
          .EXIT E1 .IFF BOOLA .EQ BOOLB;
 '        LOOP BODY';
          .EXIT E2 .IFF .DONE REPTLIM .TIMES;
 '        LOOP BODY';
 .REPEAT
          .SIT E1 .CAUSES '      GOOD EXIT CODE'; .ENDS;
          .SIT E2 .CAUSES '      TOO MANY REPEATS'; .ENDS;
 .ENDL;

 '/ END OF ZAHN LOOP';




 '/ SELECTIVE CASE STATEMENT';

 .BLOCK .WITH .EXITS CA,CB,CC,DF;
          .EXIT CA .IFF VAR1 .NE VAR2;
          .EXIT CB .IFF VAR2 .GT #40;
          .EXIT CC .IFF VAR3 .ZERO;
          .EXIT DF;
 .EPILOG
          .SIT CA .CAUSES '      ACTIONA'; .ENDS;
          .SIT CB .CAUSES '      ACTIONB'; .ENDS;
          .SIT CC .CAUSES '      ACTIONC'; .ENDS;
          .SIT DF .CAUSES '      DEFAULT'; .ENDS;
 .ENDB;

 '/ END OF SELECTIVE-CASE STATEMENT';





 '/ INTEGER DRIVEN CASE STATEMENT ';

 .BLOCK
          .EXIT .CASE INTVAL .OF C1,C2,C3 .DEFAULT CD;
 .EPILOG
          .SIT C1 .CAUSES '      ACTION1'; .ENDS;
          .SIT C2 .CAUSES '      ACTION2'; .ENDS;
          .SIT C3 .CAUSES '      ACTION3'; .ENDS;
          .SIT CD .CAUSES '      DEFAULT'; .ENDS;
 .ENDB;

 '/ END OF INTEGER DRIVEN CASE STATEMENT';





 '/ IF-THEN-ELSE';
 '/       SHOWS UNARY AND BINARY BOOLEAN OPERATORS';

 .IF (VAR1 .EQ #77) .OR
     (VAR2 .LE 999) .AND
     (VAR3 .NOTZERO)
        .THEN
     '        THEN CODE BODY';
        .ELSE
     '        ELSE CODE BODY';
 .ENDI;

 '/ END OF IF-THEN-ELSE';




 .ENDP PROGNAME;

 ****

```

figure 3.2   Example DRIL Prog Source



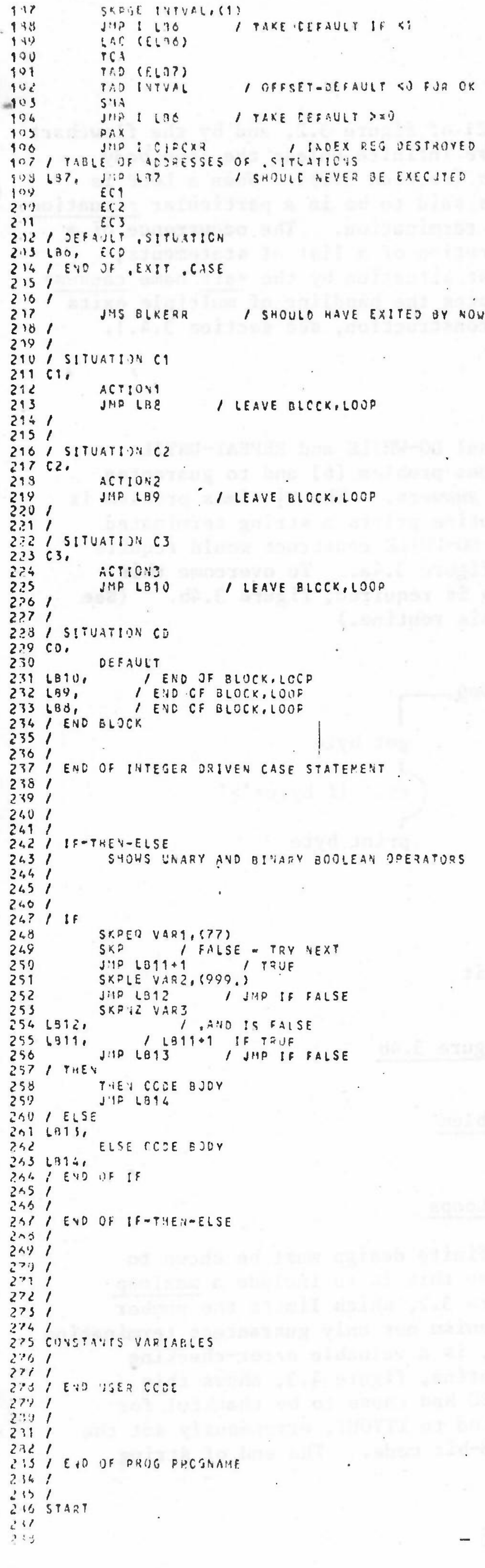
```
                  :GSIN21,META

#LISTING OF :GSIN21,ALLOBJ(5/)  PRODUCED ON 21NOV75 AT 18.30.11

  OUTPUT ON SRC 64 G832V3   UNIT U15 BY JOB ':GSIN21,META' ON 21NOV75 AT 18.40.59

DOCUMENT    ALLOBJ

     /OBJ:DRIL PROGNAME   OBJECT PROGNAME
     /
     /
     /
     /
     /
     /
     /::::::::::::::::::::::::::::::::::::::::::::::::::::
     /
     /
     /   DRIL MACRO DEFINITIONS
     /
     ,INSERT ROB:DRIL MACROS
     /
     /   END OF DRIL MACROS
     /
     /
     /::::::::::::::::::::::::::::::::::::::::::::::::::::
     /
     /
     /
     /
     /
     /
     /::::::::::::::::::::::::::::::::::::::::::::::::::::
     /
     /
     / DRIL FATAL ERROR HANDLER
     /
     /
     / INSERT FATAL ERROR HANDLER CODE
     ,INSERT ROB:DRIL FATAL
     /
     /
     / FATAL ERRORS ARE:-
     /
     /
     / FAILURE TO EXIT FROM BLOCK
    BLKERR,    FATAL (DRIL)-HIT BLOCK BOTTOM AT=,BLKERR
     /
     /
     /
     /
     /
     / END OF FATAL ERROR HANDLER
     /
     /
     /::::::::::::::::::::::::::::::::::::::::::::::::::::
     /
     /
     /
     / BEGIN USER CODE
     /
     /
     /
     /
     /
     /
     /
     /
     /
     / ZAHN LOOP - WITH GUARANTEED TERMINATION
     /
     /
     /
     / SET LOOP LIMIT COUNTER VARIABLE
           LAC (779.)
           TCA
           DAC #REPTLIM
     /
     /
     / START LOOP
    LB1,
           LOOP BODY
     /
     /
     / EXIT?
           SKPEQ BOOLA,BOOLB
           SKP       / SKP IF FALSE
           JMP E1        / EXIT E1
     /
           LOOP BODY
     /
     /
     / EXIT?
     /   STEP + TEST LOOP LIMIT COUNTER, (AND EXIT?)
```

figure 3.3   Example DRIL Prog Object



```
        ISZ REPTLIM
        SKP         / SKP IF FALSE
        JMP E2          / EXIT E2
/
        LOOP BODY
/
/
/ REPEAT LOOP
        JMP LB1
/
/
/ SITUATION E1
E1,
        GOOD EXIT CODE
        JMP LB2         / LEAVE BLOCK,LOOP
/
/
/ SITUATION E2
E2,
        TOO MANY REPEATS
LB2,        / END OF BLOCK,LOOP
/ END LOOP
/
/
/ END OF ZAHN LOOP
/
/
/
/
/ SELECTIVE CASE STATEMENT
/
/
/
/ START BLOCK
/       ,WITH   ,EXITS CA, CB, CC, DF
/
/
/ EXIT?
        SKPNE VAR1,VAR2
        SKP         / SKP IF FALSE
        JMP CA          / EXIT CA
/
/
/
/ EXIT?
        SKPGT VAR2,(40)
        SKP         / SKP IF FALSE
        JMP CB          / EXIT CB
/
/
/
/ EXIT?
        SKPZE VAR3
        SKP         / SKP IF FALSE
        JMP CC          / EXIY CC
/
        JMP DF          / EXIT DF
/
        JMS BLKERR          / SHOULD HAVE EXITED BY NOW
/
/
/ SITUATION CA
CA,
        ACTIONA
        JMP LB3         / LEAVE BLOCK,LOOP
/
/
/ SITUATION CB
CB,
        ACTIONB
        JMP LB4         / LEAVE BLOCK,LOOP
/
/
/ SITUATION CC
CC,
        ACTIONC
        JMP LB5         / LEAVE BLOCK,LOOP
/
/
/ SITUATION DF
DF,
        DEFAULT
LB5,        / END OF BLOCK,LOOP
LB4,        / END OF BLOCK,LOOP
LB3,        / END OF BLOCK,LOOP
/ END BLOCK
/
/
/ END OF SELECTIVE CASE STATEMENT
/
/
/
/
/
/ INTEGER DRIVEN CASE STATEMENT
/
/
/
/ START BLOCK
/
/
/ ,CASE INTVAL ,OF
```

figure 3.3 cont'd



```
        SKPGE INTVAL,(1)
        JMP I LB6       / TAKE DEFAULT IF <1
        LAC (EL86)
        TCA
        TAD (EL87)
        TAD INTVAL       / OFFSET-DEFAULT <0 FOR OK
        SMA
        JMP I LB6       / TAKE DEFAULT >=0
        PAX
        JMP I C1PCXR            / INDEX REG DESTROYED BY CASE
/ TABLE OF ADDRESSES OF ,SITUATIONS
LB7,    JMP LB7         / SHOULD NEVER BE EXECUTED
        EC1
        EC2
        EC3
/ DEFAULT ,SITUATION
LB6,    ECD
/ END OF ,EXIT ,CASE
/
/
        JMS BLKERR       / SHOULD HAVE EXITED BY NOW
/
/
/ SITUATION C1
C1,
        ACTION1
        JMP LB8        / LEAVE BLOCK,LOOP
/
/
/ SITUATION C2
C2,
        ACTION2
        JMP LB9        / LEAVE BLOCK,LOOP
/
/
/ SITUATION C3
C3,
        ACTION3
        JMP LB10        / LEAVE BLOCK,LOOP
/
/
/ SITUATION CD
CD,
        DEFAULT
LB10,        / END OF BLOCK,LOOP
LB9,       / END OF BLOCK,LOOP
LB8,       / END OF BLOCK,LOOP
/ END BLOCK
/
/
/ END OF INTEGER DRIVEN CASE STATEMENT
/
/
/
/
/ IF-THEN-ELSE
/        SHOWS UNARY AND BINARY BOOLEAN OPERATORS
/
/
/
/ IF
        SKPEQ VAR1,(77)
        SKP        / FALSE - TRY NEXT
        JMP LB11+1        / TRUE
        SKPLE VAR2,(999.)
        JMP LB12        / JMP IF FALSE
        SKPNZ VAR3
LB12,           / ,AND IS FALSE
LB11,        / LB11+1  IF TRUE
        JMP LB13        / JMP IF FALSE
/ THEN
        THEN CODE BODY
        JMP LB14
/ ELSE
LB13,
        ELSE CODE BODY
LB14,
/ END OF IF
/
/
/ END OF IF-THEN-ELSE
/
/
/
/
/
/
/
CONSTANTS VARIABLES
/
/
/ END USER CODE
/
/
/
/
/ END OF PROG PROGNAME
/
/
START
```

figure 3.3 cont'd

3.4 LOOP

The LOOP is illustrated by lines 7-21 of figure 3.2, and by the flowchart equivalent in figure 3.1. Loops are infinite unless the loop body contains one or more exit statements (section 3.7). When a LOOP is terminated by an exit the program is said to be in a particular situation determined by which exit caused the termination. The occurrence of a particular situation causes the execution of a list of statements, the epilog code, associated with that situation by the <sit name causes> statement. This mechanism facilitates the handling of multiple exits from a LOOP and is called the Zahn construction, see section 3.4.1.

3.4.1 The $N_{\frac{1}{2}}$ Times Problem

Why does DRIL not use the conventional DO-WHILE and REPEAT-UNTIL statements? To cope with the $N_{\frac{1}{2}}$ times problem [6] and to guarantee termination (section 3.4.2) are the answers. The $N_{\frac{1}{2}}$ times problem is illustrated by figure 3.4. The routine prints a string terminated by '>' on the teletype. To use a DO-WHILE construct would require repetition of the 'get byte' code, figure 3.4a. To overcome this overhead a more general form of loop is required, figure 3.4b. (See section 4 for the DRIL version of this routine.)

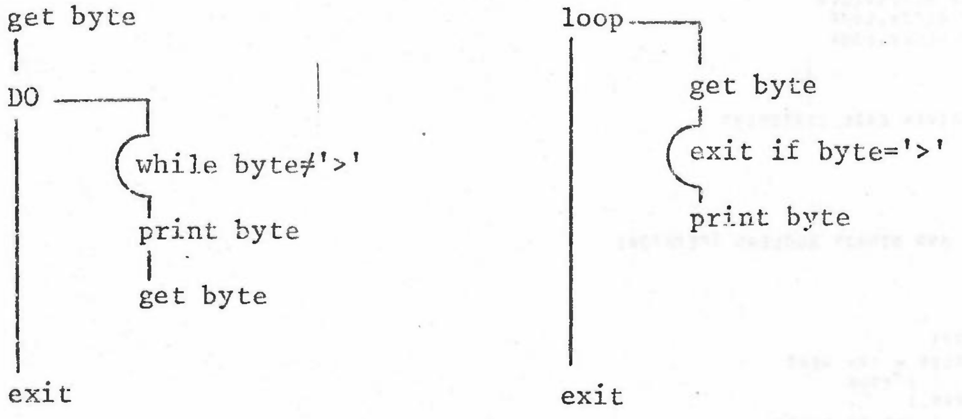


figure 3.4a

figure 3.4b

$N_{\frac{1}{2}}$ Times Problem

3.4.2 Proof of Termination of Zahn Loops

In a reliable program, all loops of finite design must be shown to terminate. The only way to guarantee this is to include a maxloop construction, lines 9 and 14 of figure 3.2, which limits the number of iterations performed. This mechanism not only guarantees termination but, with sensible repetition maxima, is a valuable error-checking facility. The teletype printing routine, figure 4.3, shows this reliable loop construction, and SYSLOG had cause to be thankful for it. An amendment to SYSLOG, unrelated to TTYOUT, erroneously set the .ASCII operator to 6-bit instead of 8-bit mode. The end of string was not detected as GTSIGN held an impossible value. Instead of looping infinitely, wearing out the teletype, SYSLOG stopped gracefully after 72 characters and printed out a meaningful error message. The error was spotted immediately because the only way a valid string could cause such an error was by TTYOUT failing to recognise '>'.

The actual SYSLOG version of TTYOUT looked like figure 3.5.

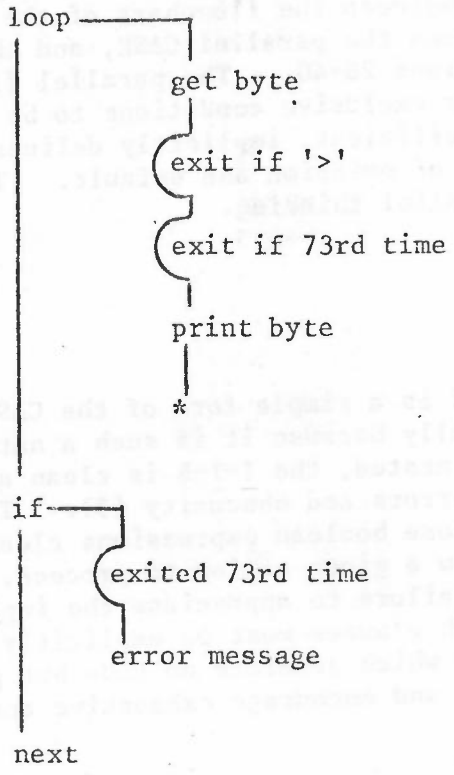


figure 3.5 SYSLOG Loop Construction

Figure 3.5 shows how the general LOOP allows multiple loop exits (impossible with DO-WHILE), but it is inefficient and tedious to follow each loop with a CASE statement to determine the exact cause of termination. The Zahn loop [4,5,6] is a construction in which the CASE statement is made an integral part of the LOOP, forming a loop epilog. The flowcharting symbol '*' now defines the end of the loop body and the start of the epilog CASE statement, see section 3.10 and figure 3.1. The Zahn loop greatly simplifies the representation of a reliable loop and this encourages its use. DRIL insists on an epilog for every loop. See figure 3.2 for the source code form and figure 3.3 for the FR80 Assembler implementation.

3.5 Block & Case

A block is a list of statements which are executed once. Blocks are simplified loops and must have exit statement(s) and epilog code. A fatal run-time error occurs if control reaches the epilog word. A major use of the conditional block is to program the selective and integer CASE statements. These are shown in lines 26-40 and lines 46-57 respectively in the example program, figure 3.2.

Note the difference between the flowchart of the CASE statement, figure 3.1, which shows the parallel CASE, and the serial selective CASE [3] encoded in lines 26-40. The parallel flowchart forces the complete and mutually exclusive conditions to be specified. The DRIL coding, though efficient, implicitly defines the cases and is more prone to errors of omission and default. This neatly demonstrates the advantage of parallel thinking.

3.6 IF-THEN-ELSE

Although IF-THEN-ELSE is a simple form of the CASE statement, it is implemented specifically because it is such a natural mode of expression. As long as it is not nested, the I-T-E is clean and easy to understand. Nested I-T-Es breed errors and obscurity [3]. They should be replaced by CASE statements whose boolean expressions clearly define the conditions which allow a given action to proceed. The IF-THEN form is not allowed as a failure to appreciate the implied ELSE can lead to errors. Null ELSE clauses must be explicitly coded using the null or ok statements which generate no code but give greater clarity to the source program and encourage exhaustive analysis. See figure 3.2, lines 26-33.

3.7 Exit

The EXIT statement causes loop termination and the execution of its associated epilog code. There are four variants of this statement.

1. EXIT <situation name>;
2. EXIT <sitname> IFF <boolean exp true>;
3. EXIT <sitname> IFF DONE <var> TIMES;
4. EXIT CASE <INTVAR> OF <sitname1>,<sitname2>,...DEFAULT <sitname>;

This comprehensive range should encourage the construction of reliable loops whose termination is guaranteed, and of CASE statements with clear and comprehensive boolean expressions governing their actions.

3.8 String

FR80 Assembler directives and statements, including comments, are enclosed by string quotes. The reason for the quotes is that they allow free format input of DRIL programs. The advantages of indenting source statements are well known. Careful use of the space character will produce a neat DRIL source program and a neat object program, see figure 3.1, 3.2, 4.1, 4.2.

3.9 DRIL Summary

Figure 3.9 is an informal summary of the present DRIL statements. Reserved words are typed as in Algol but are input as <dot><upper case reserved word>.

```
prog <ID>;

   maxloop <ID>:=<num>;

   loop

       exit <ID>;

       exit <ID> iff <BOOLEXP>;

       exit <ID> iff done <ID> times;

   repeat

       sit <ID> causes <LIST OF STATEMENTS> ends;

   endl;

   '/THIS IS A COMMENT';

   ' LAC VAR / FR80 ASM STATEMENT';

   block with exits <LIST OF IDS>;

      exit case <ID> of <LIST OF IDS> default <ID>;

      null;

      ok;

   epilog

      sit <ID> causes ok;ends;

   endb;

   if <BOOLEXP> then <LIST OF STATEMENTS>

                else <LIST OF STATEMENTS> endi;

endp <ID>;
```

figure 3.6

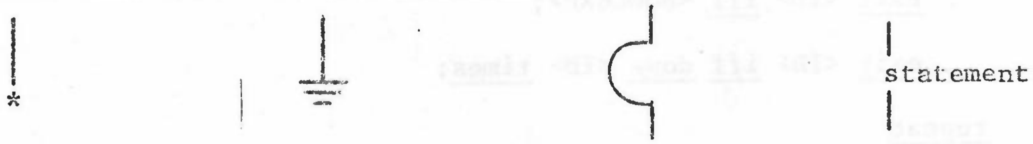
3.9.1 DRIL Boolean Operators

binary	unary
gt lt	zero
ge le	nonzero
eg ne	
and	
or	
()	

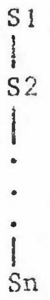
<decimal number> ::= <string of digits>

<octal number> ::= #<string of digits 0-7>

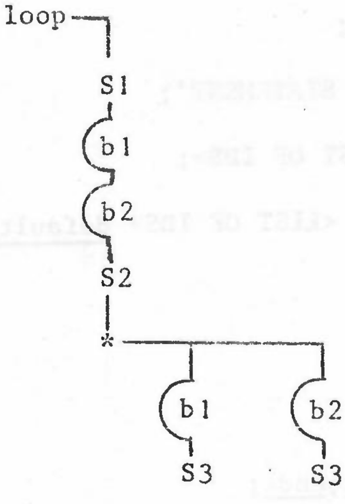
3.10 Flowchart Equivalents of DRIL Statements



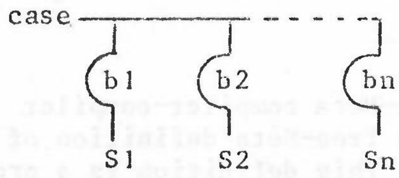
repetition termination conditional exit action



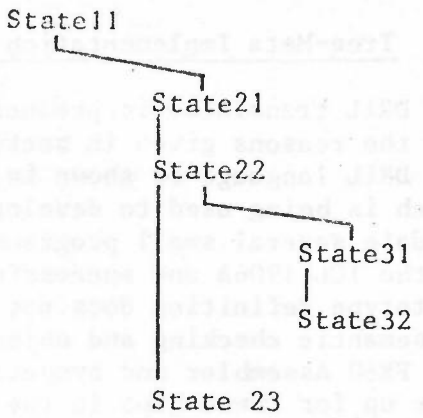
Sequence



Zahn loop



CASE/IF



REFINEMENT

4. TREE-META PRODUCED DRIL TRANSLATOR

4.1 Tree-Meta Implementation of DRIL

The DRIL translator is produced by the Tree-Meta compiler-compiler for the reasons given in section 3.1. The Tree-Meta definition of the DRIL language is shown in figure 4.6. This definition is a prototype which is being used to develop DRIL and the cross-compilation system. To date several small programs have been written in DRIL, translated in the ICL 1906A and successfully moved to and run on the FR80. The prototype definition does not take advantage of such Tree-Meta features as semantic checking and object code optimisation. The checking by the FR80 Assembler and sympathetic use of the source language respectively make up for these gaps in the implementation, which can be plugged at a later date. Priority has been given to demonstrating the feasibility of the DRIL philosophy advocated in this paper.

4.2 TTYOUT - A Working Example

TTYOUT is an FR80 routine to print a string of characters on the teletype. Hopefully this example demonstrates all the points mentioned in this paper, and in particular

1. the harmony between the Logic Design, figure 4.1, the Physical Construction, figure 4.2, and the DRIL source program, figure 4.3, plus the smooth transition through to the DRIL object program, figures 4.4, 4.5, which should resemble a hand-written program. The smallness of the example makes the Physical Construction seem rather artificial.

2. how the refinement in the Logic Diagram is 'folded' to produce the source and object programs' comments, lines 24, 26, 36 of figure 4.3 and lines 98, 100, 122 of figure 4.5.

3. that this module is an example of the N½ times problem (section 3.4.1).

4. how every statement and module has one entry point and one exit point.

5. that all DRIL loops are Zahn loops, which make error handling and proof of termination easy.

6. how the number of loop repetitions is governed by lines 12, 22 figure 4.3

7. that the loop can be proved to terminate, and hence the module itself can be shown to always exit correctly.

8. how the free format of DRIL source programs allows indentation of loop bodies, lines 14-46 of figure 4.3. Compare the source and object programs; see how the free format and the string statement are used to produce a neat object program as well as a clear source program.

9. how much clearer is the source program's IF-THEN-ELSE (lines 27-34, figure 4.3) than its assembler equivalent at lines 103-119 of figure 4.5.

10. how source comments are passed to the object program.

11. how the exit statement and its associated situation are implemented.

12. how assembler statements are passed from source to object programs via the string statement.

With the aid of the Tree-Meta manual [9] it should not be too difficult to follow the translation from source (figure 4.3) to object code (figure 4.5) according to the DRIL definition (figure 4.6).

4.3 Output from the DRIL Translator

The DRIL translator is run by a George Job Description, parameters to which allow the various output listings to be on paper or microfiche. A large DRIL program can generate over 100 pages of source program, object program (figure 4.4, 4.5), DRIL macros, fatal error handler and current Tree-Meta definition listings, all of which occupies less than half a microfiche. Microfiche are the key to a manageable project history (section 5.2).

The object program file has appended to it the source program, the DRIL macros and fatal error handler in such a format that RET's program [10] can be used to create an FR80 readable magnetic tape with each of these items as a separate file. Creation of such a tape is parametrically controlled in the Job Description.

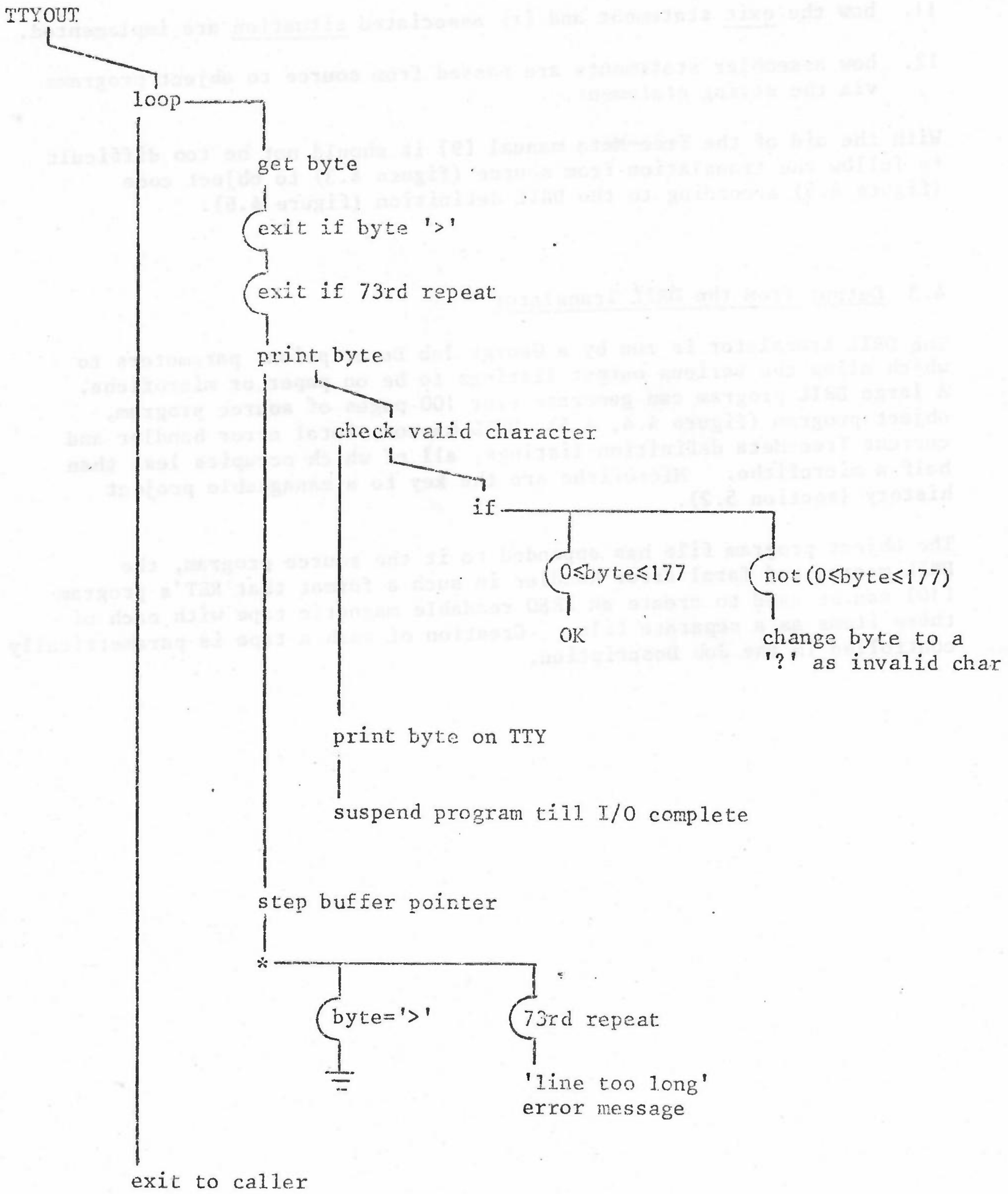


figure 4.1 TTYOUT Logic Diagram

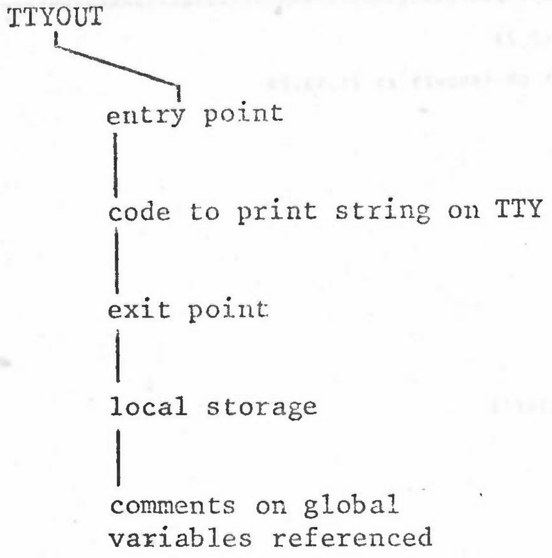

TTYOUT
entry point
code to print string on TTY
exit point
local storage
comments on global variables referenced

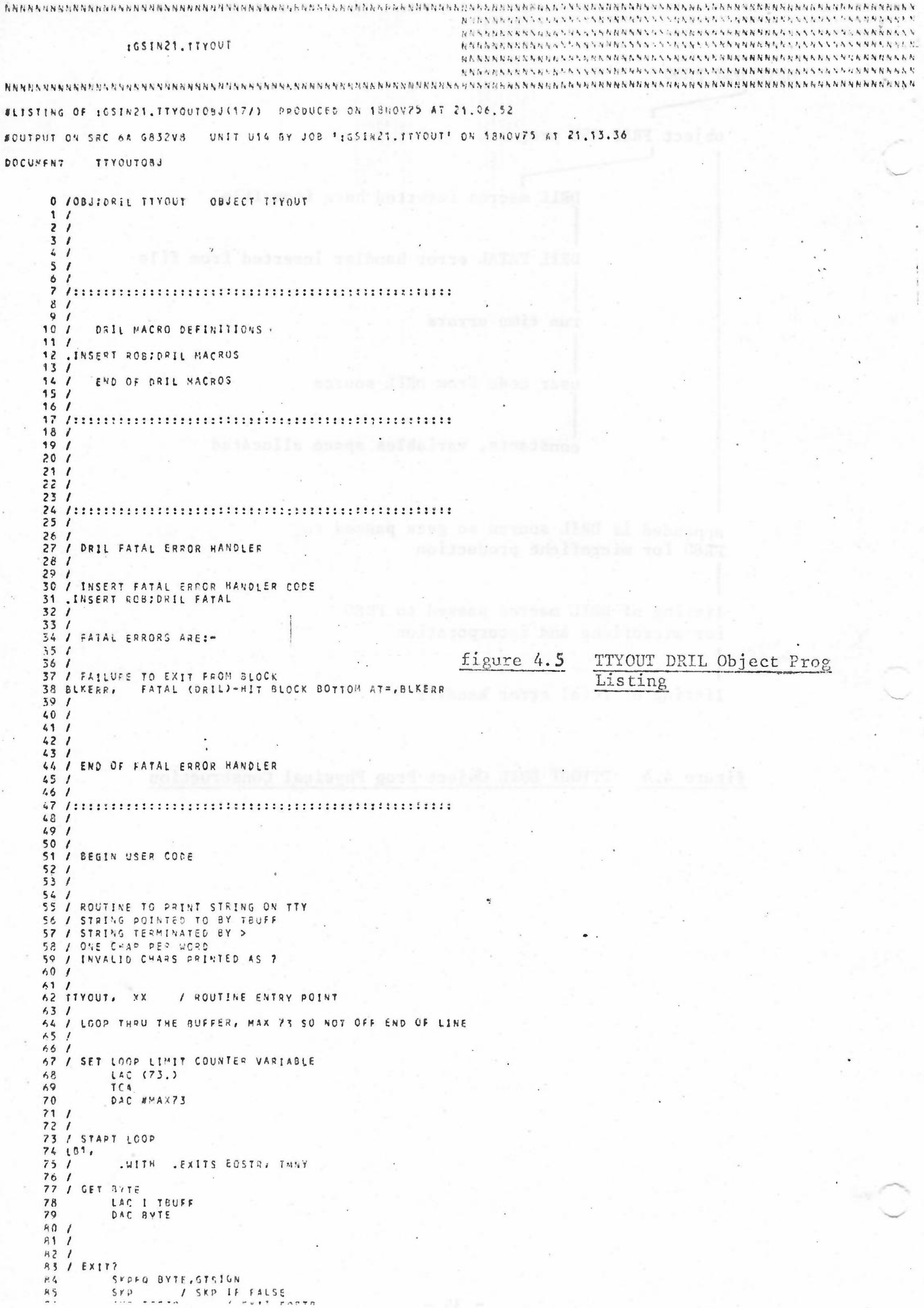
figure 4.2 TTYOUT Physical Construction

```
                 :GSIN21.TTYOUT

#LISTING OF :GSIN21.TTYOUT(9/)  PRODUCED ON 17NOV75 AT 13.42.23

#OUTPUT ON SRC 6A G932V8   UNIT U14 BY JOB ':GSIN21.TTYOUT' ON 18NOV75 AT 21.13.28

DOCUMENT    TTYOUT

 .PROG TTYOUT;

 '/ ROUTINE TO PRINT STRING ON TTY';
 '/ STRING POINTED TO BY TBUFF';
 '/ STRING TERMINATED BY > ';
 '/ ONE CHAR PER WORD';
 '/ INVALID CHARS PRINTED AS ? ';


 'TTYOUT,  XX    / ROUTINE ENTRY POINT';

 '/ LOOP THRU THE BUFFER, MAX 73 SO NOT OFF END OF LINE';
 .MAXLOOP MAX73 := 73;
 .LOOP
         .WITH .EXITS EOSTR,TMNY;

         '/ GET BYTE';
 '       LAC I TBUFF';
 '       DAC BYTE';

         .EXIT EOSTR .IFF BYTE .EQ GTSIGN;

         .EXIT TMNY .IFF .DONE MAX73 .TIMES;

         '/ PRINT BYTE';

         '/ CHECK VALID CHAR';
         .IF (BYTE .GT #177) .OR (BYTE .LT 0)
             .THEN
                 '/ CHANGE TO ? ';
         '       LAC QMARK';
         '       DAC BYTE';
             .ELSE
                 .OK;
         .ENDI;

         '/ PRINT ON TTY';
 '       LAC BYTE';
 '       TLS';

         '/ SUSPEND UNTIL I/O FINISHED';
 '       TSF';
 '       JMP .-1';

         '/ STEP POINTER';
 '       ISZ TBUFF';
 '       NOP';

 .REPEAT
     .SIT EOSTR .CAUSES .OK; .ENDS;
     .SIT TMNY  .CAUSES '      JMS TOOLNG'; .ENDS;
 .ENDL;

 '       JMP I TTYOUT    / EXIT TO CALLER';



 '/ LOCAL STORAGE';
     'BYTE,       0';
     'GTSIGN,     .ASCII/>/ ';
     'QMRK,       .ASCII/?/ ';


 '/ GLOBAL REFERENCES';
 '/   TBUFF    - POINTS TO START OF STRING, CHANGED BY THIS ROUTINE';
 '/   TOOLNG   - PRINTS ERROR MESSAGE, LINE TOO LONG';


 .ENDP TTYOUT;

 ****
```

figure 4.3 TTYOUT DRIL Source Prog Listing

- DRIL OBJ PROG
    - object FR80 ASM prog
        - DRIL macros inserted here from file
        - DRIL FATAL error handler inserted from file
        - run time errors
        - user code from DRIL source
        - constants, variables space allocated
    - appended is DRIL source so gets passed to FR80 for microfiche production
    - listing of DRIL macros passed to FR80 for microfiche and incorporation
    - listing of fatal error handler

figure 4.4 TTYOUT DRIL Object Prog Physical Construction

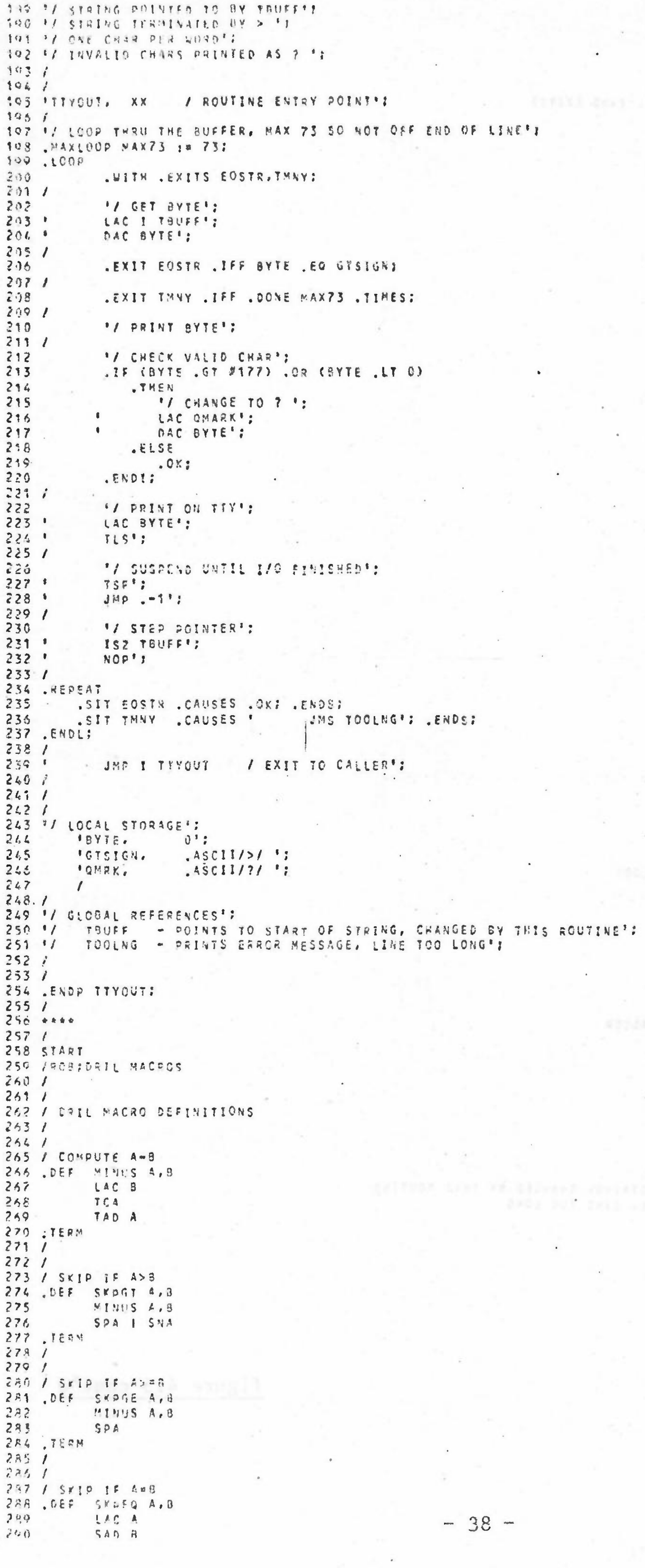


```
        :GSIN21.TTYOUT

#LISTING OF :GSIN21.TTYOUTOBJ(17/)  PRODUCED ON 18NOV75 AT 21.06.52

#OUTPUT ON SRC 6A G832V8   UNIT U14 BY JOB ':GSIN21.TTYOUT' ON 18NOV75 AT 21.13.36

DOCUMENT    TTYOUTOBJ

/OBJ:DRIL TTYOUT   OBJECT TTYOUT
/
/
/
/
/
/
/::::::::::::::::::::::::::::::::::::::::::::::::::
/
/
/   DRIL MACRO DEFINITIONS
/
.INSERT ROB:DRIL MACROS
/
/   END OF DRIL MACROS
/
/
/::::::::::::::::::::::::::::::::::::::::::::::::::
/
/
/
/
/
/
/::::::::::::::::::::::::::::::::::::::::::::::::::
/
/
/ DRIL FATAL ERROR HANDLER
/
/
/ INSERT FATAL ERROR HANDLER CODE
.INSERT ROB:DRIL FATAL
/
/
/ FATAL ERRORS ARE:-
/
/
/ FAILURE TO EXIT FROM BLOCK
BLKERR,   FATAL (DRIL)-HIT BLOCK BOTTOM AT=,BLKERR
/
/
/
/
/
/ END OF FATAL ERROR HANDLER
/
/
/::::::::::::::::::::::::::::::::::::::::::::::::::
/
/
/
/ BEGIN USER CODE
/
/
/
/ ROUTINE TO PRINT STRING ON TTY
/ STRING POINTED TO BY TBUFF
/ STRING TERMINATED BY >
/ ONE CHAR PER WORD
/ INVALID CHARS PRINTED AS ?
/
/
TTYOUT,  XX     / ROUTINE ENTRY POINT
/
/ LOOP THRU THE BUFFER, MAX 73 SO NOT OFF END OF LINE
/
/
/ SET LOOP LIMIT COUNTER VARIABLE
        LAC (73.)
        TCA
        DAC #MAX73
/
/
/ START LOOP
LB1,
/        .WITH  .EXITS EOSTR, TMNY
/
/ GET BYTE
        LAC I TBUFF
        DAC BYTE
/
/
/
/ EXIT?
        SKPEQ BYTE,GTSIGN
        SKP        / SKP IF FALSE
```

figure 4.5 TTYOUT DRIL Object Prog Listing

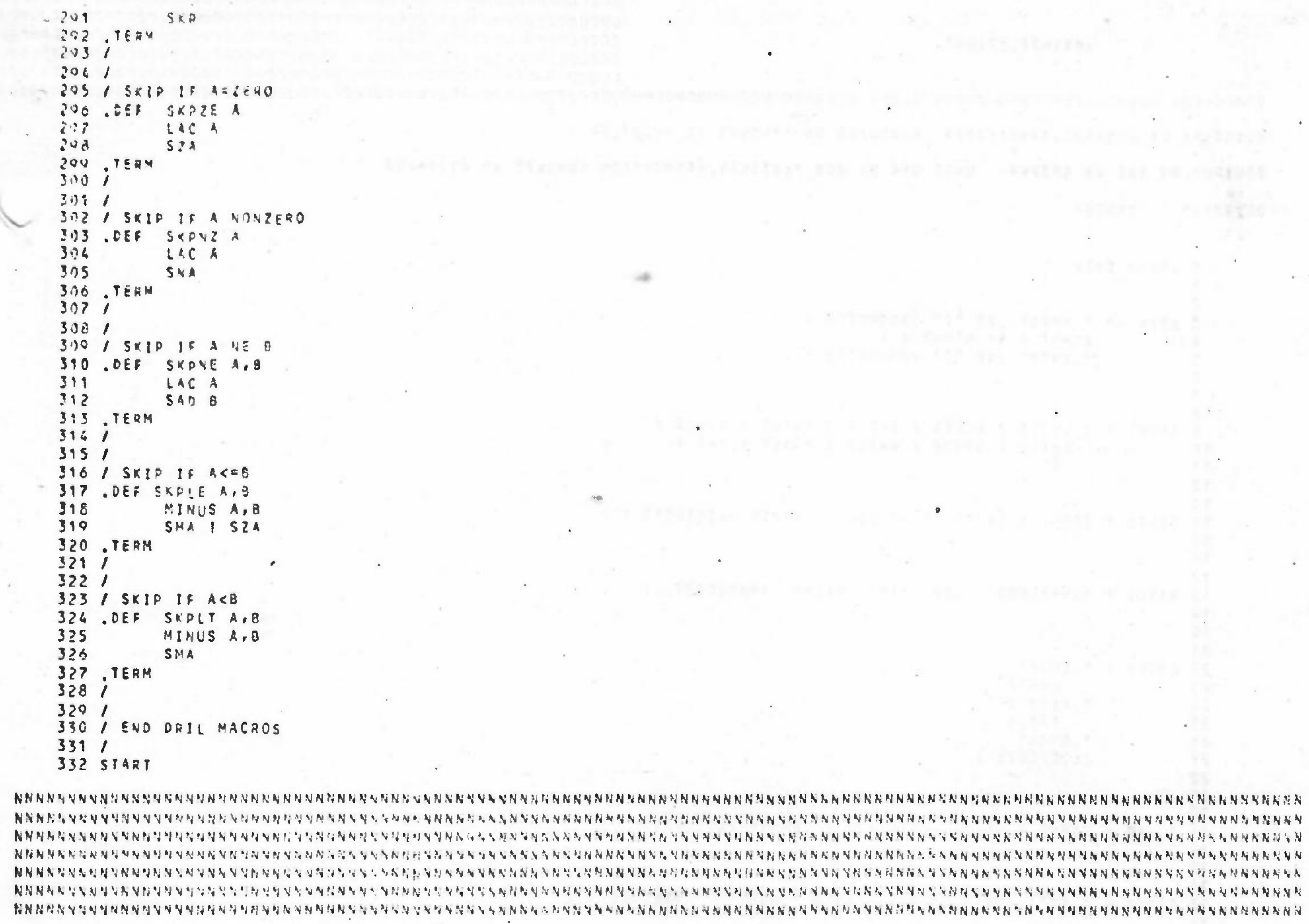
```
/
/
/
/
/ EXIT?
/    STEP + TEST LOOP LIMIT COUNTER, (AND EXIT?)
        ISZ MAX73
        SKP          / SKP IF FALSE
        JMP TMNY          / EXIT TMNY
/
/
/ PRINT BYTE
/
/ CHECK VALID CHAR
/
/
/ IF
        SKPGT BYTE,(177)
        SKP          / FALSE - TRY NEXT
        JMP LB2+1          / TRUE
        SKPLT BYTF,(0.)
LB2,         / LB2+1   IF TRUE
        JMP LB3          / JMP IF FALSE
/ THEN
/ CHANGE TO ?
        LAC QMARK
        DAC BYTE
        JMP LB4
/ ELSE
LB3,
/         OK,NULL STATEMENT
LB4,
/ END OF IF
/
/
/ PRINT ON TTY
        LAC BYTE
        TLS
/
/ SUSPEND UNTIL I/O FINISHED
        TSF
        JMP .-1
/
/ STEP POINTER
        ISZ TBUFF
        NOP
/
/
/
/ REPEAT LOOP
        JMP LB1
/
/
/ SITUATION EOSTR
EOSTR,
/         OK,NULL STATEMENT
        JMP LB5          ./ LEAVE BLOCK,LOOP
/
/
/ SITUATION TMNY
TMNY,
        JMS TOOLNG
LB5,         / END OF BLOCK,LOOP
/ END LOOP
/
/
        JMP I TTYOUT       / EXIT TO CALLER
/
/
/
/ LOCAL STORAGE
BYTE,        0
GTSIGN,      .ASCII/>/
QMRK,        .ASCII/?/
/
/
/ GLOBAL REFERENCES
/    TBUFF    - PCINTS TO START OF STRING, CHANGED BY THIS ROUTINE
/    TOOLNG   - PRINTS ERROR MESSAGE, LINE TOO LONG
/
/
/
/
/
CONSTANTS VARIABLES
/
/
/ END USER CODE
/
/
/
/
/ END OF PROG TTYOUT
/
/
START
/SRC;DRIL TTYOUT  SOURCE TTYOUT
/
/
.PROG TTYCUT;
/
"/ ROUTINE TO PRINT STRING ON TTY";
```

figure 4.5 cont'd



```
'/ STRING POINTED TO BY TBUFF';
'/ STRING TERMINATED BY > ';
'/ ONE CHAR PER WORD';
'/ INVALID CHARS PRINTED AS ? ';
/
/
'TTYOUT,  XX     / ROUTINE ENTRY POINT';
/
'/ LOOP THRU THE BUFFER, MAX 73 SO NOT OFF END OF LINE';
.MAXLOOP MAX73 := 73;
.LOOP
        .WITH .EXITS EOSTR,TMNY;
/
        '/ GET BYTE';
'       LAC I TBUFF';
'       DAC BYTE';
/
        .EXIT EOSTR .IFF BYTE .EQ GTSIGN;
/
        .EXIT TMNY .IFF .DONE MAX73 .TIMES;
/
        '/ PRINT BYTE';
/
        '/ CHECK VALID CHAR';
        .IF (BYTE .GT #177) .OR (BYTE .LT 0)
           .THEN
              '/ CHANGE TO ? ';
       '      LAC QMARK';
       '      DAC BYTE';
           .ELSE
              .OK;
        .ENDI;
/
        '/ PRINT ON TTY';
'       LAC BYTE';
'       TLS';
/
        '/ SUSPEND UNTIL I/O FINISHED';
'       TSF';
'       JMP .-1';
/
        '/ STEP POINTER';
'       ISZ TBUFF';
'       NOP';
/
.REPEAT
    .SIT EOSTR .CAUSES .OK; .ENDS;
    .SIT TMNY  .CAUSES '      JMS TOOLNG'; .ENDS;
.ENDL;
/
'       JMP I TTYOUT    / EXIT TO CALLER';
/
/
/
'/ LOCAL STORAGE';
    'BYTE,        0';
    'GTSIGN,      .ASCII/>/ ';
    'QMRK,        .ASCII/?/ ';
    /
/
'/ GLOBAL REFERENCES';
'/   TBUFF   - POINTS TO START OF STRING, CHANGED BY THIS ROUTINE';
'/   TOOLNG  - PRINTS ERROR MESSAGE, LINE TOO LONG';
/
/
.ENDP TTYOUT;
/
****
/
START
/ROB;DRIL MACROS
/
/
/ DRIL MACRO DEFINITIONS
/
/
/ COMPUTE A-B
.DEF  MINUS A,B
       LAC B
       TCA
       TAD A
.TERM
/
/
/ SKIP IF A>B
.DEF  SKPGT A,B
       MINUS A,B
       SPA ! SNA
.TERM
/
/
/ SKIP IF A>=B
.DEF  SKPGE A,B
       MINUS A,B
       SPA
.TERM
/
/
/ SKIP IF A=B
.DEF  SKPEQ A,B
       LAC A
       SAD B
```

figure 4.5 cont'd



```
        SKP
.TERM
/
/
/ SKIP IF A=ZERO
.DEF    SKPZE A
        LAC A
        SZA
.TERM
/
/
/ SKIP IF A NONZERO
.DEF    SKPNZ A
        LAC A
        SNA
.TERM
/
/
/ SKIP IF A NE B
.DEF    SKPNE A,B
        LAC A
        SAD B
.TERM
/
/
/ SKIP IF A<=B
.DEF SKPLE A,B
        MINUS A,B
        SMA ! SZA
.TERM
/
/
/ SKIP IF A<B
.DEF    SKPLT A,B
        MINUS A,B
        SMA
.TERM
/
/
/ END DRIL MACROS
/
START

NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
```

figure 4.5 cont'd

```
:GSIN21.TTYOUT

#LISTING OF :GSIN21.TRNDEF(8/)  PRODUCED ON 18NOV75 AT 16.27.59

#OUTPUT ON SRC 6A G832V8   UNIT U14 BY JOB ':GSIN21.TTYOUT' ON 18NOV75 AT 21.14.08

DOCUMENT    TRNDEF

.META DRIL


DRIL  = '.PROG' .ID ':' :FNAMC[1] *
         STMNT * S( STMNT * )
         '.ENDP' .ID ':' :ENAMC[1] * ;



STMNT = ( LOOPS / BLCKS / IFTES / FR8OS / NULLS /
          EXITS / SITUS / UNTLS / MAXLS ) ';' ;




LISTS = STMNT ( LISTS :LISTC[2] / .EMPTY :LISTC[1] ) ;



MAXLS = '.MAXLOOP'  .ID  ':='  PRIMS  :MAXLC[2]  ;



LOOPS = '.LOOP'
           LISTS
         '.REPEAT'
           EPILS
         '.ENDL'
         :LOOPC[2] ;



EPILS = STMNT ( EPILS :EPILC[2] / .EMPTY :EPILC[1] ) ;



SITUS = '.SIT' .ID '.CAUSES' LISTS '.ENDS'    :SITUC[2] ;



BLCKS = '.BLOCK'
           LISTS
         '.EPILOG'
           EPILS
         '.ENDB'
         :BLCKC[2] ;



IFTES = '.IF' BEXPS '.THEN' LISTS '.ELSE' LISTS '.ENDI'  :IFTEC[3] ;

BEXPS = BANDS ( '.OR' BEXPS  :BEXPC[2] / .EMPTY  :BEXPC[1] ) ;

BANDS = CONDS ( '.AND' CONDS  :BANDC[2] / .EMPTY  :BANDC[1] ) ;

CONDS = ( PRIMS ( '.GT' PRIMS  ↑'GT'  :SKPTC[3] /
                  '.GE' PRIMS  ↑'GE'  :SKPTC[3] /
                  '.EQ' PRIMS  ↑'EQ'  :SKPTC[3] /
                  '.NE' PRIMS  ↑'NE'  :SKPTC[3] /
                  '.LE' PRIMS  ↑'LE'  :SKPTC[3] /
                  '.LT' PRIMS  ↑'LT'  :SKPTC[3] /
                  '.ZERO'             :SKPZE[1] /
                  '.NONZERO'          :SKPNZ[1] ) /
         '(' BEXPS ')'  :BEXPC[1] ) ;

PRIMS = ( .ID        :IDENC[1]  /
          .NUM       :DNUMC[1]  /
          '#' .NUM   :ONUMC[1]  )  ;



FR8OS = .SR  :FR8OC[1] ;



NULLS = ( '.NULL' / '.OK' )  ↑ 'OK,NULL STATEMENT'  :NULLC[1]   ;



EXITS = '.EXIT'
         ( .ID ( '.IFF' ( '.DONE' .ID '.TIMES'  :DONEC[1]  :EXITC[2]  /
                          BEXPS  :EXITC[2]  ) /
                .EMPTY  :EXITC[1]  )
            / ECASS :EXITC[1] )  ;

ECASS = '.CASE' .ID '.OF' SITIS '.DEFAULT' .ID  :ECASC[3] ;


```

figure 4.6   TREE-META Definition of DRIL

```

SITLS = .ID ( ',' SITLS :SITLC[2] / .EMPTY :SITLC[1] ) ;



UNTLS = ( '.UNTIL' / '.WITH'  '.EXITS' ) ELSTS   :UNTLC[1] ;



ELSTS = .ID ( ',' ELSTS :ELSTC[2] / .EMPTY :ELSTC[1] ) ;










PNAMC[-]         => '/OBJ:DRIL ' *1  '   OBJECT ' *1 % % %
                    MACRO[]
                    FATAL[]
                    ;

MACRO/           => % % % %
                    '/::::::::::::::::::::::::::::::::::::::::::::::::::'
                    % % %
                    '/   DRIL MACRO DEFINITIONS'
                    % %
                    '.INSERT ROB:DRIL MACROS'
                    % %
                    '/   END OF DRIL MACROS'
                    % % %
                    '/::::::::::::::::::::::::::::::::::::::::::::::::::'
                    % % % %
                    ;


FATAL           /=> % % %
                    '/::::::::::::::::::::::::::::::::::::::::::::::::::'
                    % % %
                    '/ DRIL FATAL ERROR HANDLER'
                    % % %
                    '/ INSERT FATAL ERROR HANDLER CODE' %
                    '.INSERT ROB:DRIL FATAL' %
                    % %
                    '/ FATAL ERRORS ARE:-' %
                    % %
                    '/ FAILURE TO EXIT FROM BLOCK' %
                    'BLKERR,   FATAL (DRIL)-HIT BLOCK BOYTOM AT=,BLKERR' %
                    % %
                    % % %
                    '/ END OF FATAL ERROR HANDLER'
                    % % %
                    '/::::::::::::::::::::::::::::::::::::::::::::::::::'
                    % % % %
                    '/ BEGIN USER CODE' % % %
                    ;


ENAMC[-]         => % % %
                    'CONSTANTS VARIABLES'
                    % % %
                    '/ END USER CODE' % %
                    % % % '/ END OF PROG ' *1 % % %
                    'START' %
                    '/SRC:DRIL ' *1  '  SOURCE '  *1 %
                    ;



LISTC[-,-]       => *1 *2
     [-]         => *1 ;



MAXLC[-,-]       => % %
                    '/ SET LOOP LIMIT COUNTER VARIABLE' %
                    '      LAC '  *2  %
                    '      TCA'  %
                    '      DAC '  '#' *1  %
                    ;

LOOPC[-,-]       => % % '/ START LOOP' %
                    #1 ',' %
                    *1
                    % % '/ REPEAT LOOP' %
                    '      JMP '  #1  %
                    *2
                    '/ END LOOP' % % ;

EPILC[-,-]       => *1
                    '      JMP '  #1  '       / LEAVE BLOCK,LOOP' %
                    *2
                    #1  ',' '       / END OF BLOCK,LOOP' %
     [-]         => *1 ;



SITUC[-,-]       => % % '/ SITUATION '  *1  %
                    *1  ','  %
```

figure 4.6 cont'd

```
                  *2 ;



BLCKC[-,-]     => % % '/ START BLOCK' %
                  *1
                  '       JMS BLVERR      / SHOULD HAVE EXITED BY NOW' %
                  *2
                  '/ END BLOCK' % % ;



IFTEC[-,-,-] => % % '/ IF ' %
                  *1
                  '       JMP '  #1  '       / JMP IF FALSE' %
                  '/ THEN' %
                  *2
                  '       JMP '  #2 %
                  '/ ELSE' %
                  #1  ','  %
                  *3
                  #2  ','  %
                  '/ END OF IF' % % ;




SEXPC[-,-]     => *1
                  '       SKP'  '       / FALSE - TRY NEXT' %
                  '       JMP '  #1  '+1'  '       / TRUE' %
                  *2
                  #1  ',      / '  #1  '+1  IF TRUE' %
      [-]      => *1 ;



BANDC[-,-]     => *1
                  '       JMP '  #1  '       / JMP IF FALSE' %
                  *2
                  #1  ',             / .AND IS FALSE' %
      [-]      => *1 ;

SKPTC[-,-,-] => '        SKP'  *3  ' '  *1  ','  *2  %  ;

SKPZE[-]       => '       SKPZE '  *1  %  ;

SKPNZ[-]       => '       SKPNZ '  *1  %  ;

IDENC[-]       => *1 ;

DNUMC[-]       => '('  *1  '.)'  ;

ONUMC[-]       => '('  *1   ')'  ;



FRBOC[-]       => *1 % ;



EXITC[-,-]     => % %
                  '/ EXIT?' %
                  *2
                  '       SKP       / SKP IF FALSE' %
                  '       JMP '  *1  '       / EXIT '  *1  % %
      [ECASC[-,-,-]] => *1 % %
      [-]      => '       JMP '  *1  '       / EXIT '  *1  % % ;

ECASC[-,-,-] => % % '/ .CASE '  *1  ' .OF'  %
                  '       SKPGE '  *1  ',(1)'  %
                  '       JMP I '  #2  '       / TAKE DEFAULT IF <1'  %
                  '       LAC ($'  #2  ')'  %
                  '       TCA'  %
                  '       TAD ($'  #1  ')'  %
                  '       TAD '  *1  '       / OFFSET-DEFAULT <0 FOR OK'  %
                  '       SMA'  %
                  '       JMP I '  #2  '       / TAKE DEFAULT >=0'  %
                  '       PAX'  %
                  '       JMP I 0IPCXR' '       / INDEX REG DESTROYED BY CASE'  %
                  '/ TABLE OF ADDRESSES OF .SITUATIONS'  %
                  #1  ',  JMP '  #1  '       / SHOULD NEVER BE EXECUTED'  %
                  *2
                  '/ DEFAULT .SITUATION'  %
                  #2  ',  $'  *3    %
                  '/ END OF .EXIT .CASE'  %  ;



DONEC[-]       => '/   STEP + TEST LOOP LIMIT COUNTER, (AND EXIT?)'  %
                  '       ISZ '  *1 %
                  ;



SITLC[-,-]     => '       $'  *1  %
                  *2
      [-]      => '       $'  *1  % ;



UNTLC[-]       => '/      .WITH  .EXITS'  *1 ;

```

figure 4.6 cont'd

```


NULLC[-]       => '/        '   *1   %   ;



ELSTC[-,-]     => ' '   *1   ','   *2
     [-]       => ' '   *1   %   ;


.END
****

000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

+**********************************************************************************************************************************+








+**********************************************************************************************************************************+
```

figure 4.6 cont'd

5. PROGRAMMING STYLE AND STANDARDS

5.1 Discussion

Style is still a major ingredient in programs because they are written by individuals. A clear, consistent style facilitates third party understanding and reliability. It is worth bearing in mind that the third party often referred to in this paper is likely to be the program's creator ten weeks after the code has been written!

A programmer's style includes his methods and his assumptions. When more than one individual works on a project these methods and assumptions must be shared by all. This is the point where a 'style' must become a 'standard' if consistency is to be maintained. Only when a good style has been developed should it be used widely and evolve into a standard. Standards should be a welcome feature of a project, being formal and helpful statements of good points, not millstones.

To help the author, and anyone else who gets involved with DRIVER, section 5.2 outlines those practices which are currently 'common law'. They form the nucleus of an evolving set of DRIVER standards. Section 5.3 is based on a book called 'The Elements of Programming Style' [1]. If the reader has any good tips to add to this list then RWW will be pleased to receive them. Hopefully the differences and similarities between 5.2 and 5.3 show the evolution from style to standards. Programming is still a craft [11]; let us encourage craftsmanship.

5.2 DRIVER Standards

1. A Project History must be kept.
   Use microfiche as much as possible.

2. A Project Diary must be kept as part of the Project History.

3. All design must be by SWR, and documented by 3-D flowcharts.
   All design decisions must be recorded in the Project History.

4. All code must be written in DRIL directly from the 3-D flowchart.
   No program is to be patched at the assembler or binary level.

5. Code and flowcharts (logical and physical) must maintain their exact correspondence. Amend all three simultaneously.

6. Flowchart refinement must be 'folded' to form source comments.

7. All current versions of programs must have a complete set of listings filed in the Project History. The set must include source, object, DRIL definition, FR80 listing assembler and cross-reference listings, preferably on microfiche.

8. All modules must have a single entry point and a single exit point.

9. All modules must declare, as comment, those global variables referenced within, and what happens to them.

10. Data, both local and global must be physically separate from the code, because of overlaying activity.

11. All code must be pure, again because of overlaying.

12. All modules must be proven to always exit whenever they are entered; therefore all loops must terminate. Prove as much as possible about the module's behaviour - document the assumptions.

13. All tests must be planned. The plan and results must be filed in the Project History.

14. Construction and testing must be by the top-down, program stub method. No test beds.

15. Write down 'the little things'. File them in the Project History so they are not forgotten.

16. All modules must be designed down to the last detail before coding commences. Good detailed design is good engineering.

5.3 Style

1. Always aim for simplicity, clarity and reliability. Gain efficiency by good design not by 'tricky' coding. Avoid the temptation to make irrelevant local optimisations. Instrument the program, determine the bottlenecks, then remove them 'hygenically'.

2. Do not be afraid to build a prototype module to gain an understanding of the problem. Learn from it, scrap it and then build the production module.

3. Design and re-design, rather than code and re-code. 'Polish' the module as an author improves a paragraph. Bad programs are easy to create; good programs are hard work.

4. Write your program as though it were going to be compiled and tested by a complete stranger.

5. Write clearly, don't be too clever.

6. Say what you mean, simply and directly.

7. Use library functions.

8. Avoid temporary variables.

9. Write clearly - don't sacrifice clarity for 'efficiency'.

10. Let the machine do the dirty work.

11. Replace repetitive expressions by calls to a common function.

12. Parenthesize to avoid ambiguity.

13. Choose variable names that won't be confused.

14. Avoid unnecessary branches.

15. Don't use conditional branches as a substitute for a logical expression.

16. If a logical expression is hard to understand, try transforming it.

17. Use data arrays to avoid repetitive control sequences.

18. Choose a data representation that makes the program simple.

19. Write first in an easy-to-understand pseudo-language; then translate into whatever language you have to use ie 3-D flowcharts → DRIL → FR80 ASM.

20. Use CASE to implement multi-way branches.

21. Modularize. Use subroutines.

22. Do not nest IF-THEN-ELSE, use a CASE.

23. Use GOTOs only to implement a fundamental structure such as LOOP.

24. Avoid GOTOs completely if you can keep the program readable.

25. Don't patch bad code - rewrite it.

26. Write and test a big program in small pieces. Plan the test. Top-down stubs method.

27. Use recursive procedures for recursively-defined data structures.

28. Test input for plausibility and validity.

29. Make sure input doesn't violate the limits of the program.

30. Terminate input by end-of-file or marker, not by count.

31. Identify bad input; recover if possible.

32. Make input easy to prepare and output self-explanatory.

33. Use uniform input formats.

34. Make input easy to proofread.

35. Use free-form input when possible.

36. Use self-identifying input. Allow defaults, Echo both on output.

37. Make sure all variables are initialized before use by executable code.

38. Don't stop at one bug.

39. Use debugging compilers.

40. Initialize constants with DATA statements or INITIAL attributes; initialize variables with executable code.

41. Watch out for off-by-one errors.

42. Take care to branch the right way on equality.

43. Prove loop termination.

44. Make sure your code 'does nothing' gracefully.

45. Test programs at their boundary values.

46. Check some answers by hand.

47. 10.0 times 0.1 is hardly ever 1.0.

48. Don't compare floating point numbers solely for equality.

49. Make it right before you make it faster.

50. Make it fail-safe before you make it faster.

51. Make it clear before you make it faster.

52. Don't sacrifice clarity for small gains in 'efficiency'.

53. Let your compiler do the simple optimizations.

54. Don't strain to re-use code; reorganise instead.

55. Make sure special cases are truly special.

56. Keep it simple to make it faster.

57. Don't diddle code to make it faster - find a better algorithm.

58. Instrument your programs. Measure before making 'efficiency' changes.

59. Make sure comments and code agree. Fold 3-D comment hierarchy into program source code.

60. Don't just echo the code with comments - make every comment count.

61. Don't comment bad code - rewrite it.

62. Use variable names that mean something.

63. Use statement labels that mean something.

64. Format a program to help the reader understand it.

65. Document your data layouts.

66. Don't over-comment.

67. Make sure every loop always terminates.

68. Prove termination of program.

69. Loops should have only one entry and one entry point.

70. Modules should have only one entry and one entry point.

6. REFERENCES

1. KERNIGHAN, PLAUSER
Elements of Programming Style
McGraw Hill

2. DONZEAU-GOUGE et al
A Structure-oriented Program Editor
International Computing Symposium 1975 ed Potier

3. WEINBERG et al
IF-THEN-ELSE Considered Harmful
Sigplan August 1975

4. ZAHN
A Control Statement for Natural Top Down Structured Programming.
Lecture Notes in Computer Science
Vol 19 Paris 1974 ed Goos, Hartmanis

5. KNUTH, ZAHN
Ill Chosen Use of 'Event'
CACM Vol 18 No 6 June 75

6. KNUTH
Structured Programming with <u>goto</u> Statements
Computing Surveys Vol 6 No 4 Dec 74

7. DIJKSTRA, DAHL, HOARE
Structured Programming
Academic Press 72

8. FR80 Discussion Paper 15, paragraph 1.9

9. HOPGOOD
The 1906A Tree-Meta Manual
ACL

10. FR80 Technical Paper 14

11. DIJKSTRA
EWD469

12. BURROUGHS CORP
B6700 CANDE Manual