KENT ON-LINE SYSTEM

Document: KUSE/UNRAVEL/1

UNRAVEL User's manual for the ICL 4130

P.J. Brown
University of Kent at Canterbury
December 1971.

Table of contents

Chapter 5 UNRAVEL on the ICL 4130

Chapter 1 Introduction

1.1 Uses of UNRAVEL

UNRAVEL is a programming language for printing out information from core store. There already exist several dumping programs that do this, so it is best to start by describing why UNRAVEL is useful.

There are two main problems with traditional dumping programs. Firstly information is printed out in a uniform format, to a uniform base (e.g. octal) and without any interpretation or annotation. The unfortunate reader of the dump has to go through a mass of information to extract what he needs. Often he has to perform tortuous conversions, e.g. from octal to decimal, character, or program address.

The second problem with conventional dumping programs is that it is often impossible to extract information that is indirectly addressed, e.g. given that location 131 points at a 40 word table, print the table, or given the start of a linked list, print all the items on the list.

The purpose of UNRAVEL is to surmount these problems by providing the user with a programming language to describe how a dump is to be made. In other words UNRAVEL is used to put "intelligence" into dumps. An UNRAVEL program can be made to interpret the material to be dumped to save the reader the trouble of doing so. For example assume a 24-bit word of information in a table describes the usage of an I/O device in the following way:

First two bits	:	state, e.g. free, busy.
Next seven bits	:	priority level.
Next fifteen bits	:	pointer to name of user.

and assume further the table contains 30 entries, corresponding to devices 0 to 29. An UNRAVEL program could be made to interpret the table accordingly and print out information in a form such as:

```
DEVICE 0 BUSY AT PRIORITY LEVEL 6.  USER IS CU/RO99
DEVICE 1  FREE
DEVICE 2  ...
```

Many of the current uses of UNRAVEL have entirely involved searching core storage for certain information and then printing out the interpretation of that information. For example in the Kent On-line System (KOS) it is quite easy to find out which sub-systems, I/O devices, disc files, etc., each console is using, and print this information out.

This leads to a further usage. UNRAVEL programs can be written to provide dynamic monitoring of core storage locations. The current contents of a storage location is remembered and the UNRAVEL program is set into a loop, comparing the current value of the storage location with the previous one and, if it has changed, printing it out. This usage is, however, mainly for the systems programmer as it may be necessary to understand scheduling algorithms if results are to be interpreted correctly.

In spite of these relatively sophisticated applications of UNRAVEL, it is expected that by far the most popular application will be in providing post-mortem dumps. Possibly it will be at its most useful in finding the kind of out-of-the-way bug that can arise in established software. An UNRAVEL dump which prints out all the tables, lists, stacks, buffers, etc. that the software uses can be invaluable. The listing should be well annotated and all information should be in a readable form. The UNRAVEL program may even be made to give its own comments on apparent errors e.g.

PRIORITY LEVEL OF DEVICE 3 IS ABOVE THE LIMIT

1.2 Implementations of UNRAVEL

UNRAVEL is not tied to any particular computer but can be implemented on almost any one.

There are, however, features of UNRAVEL which will vary between implementations for different computers and even for different operating systems on the same machine. Hence this manual has been organized as follows: the first four Chapters describe features of UNRAVEL that are common to all implementations, and the last Chapter, Chapter 5, describes machine-dependent features. Each implementation will have its own version of Chapter 5.

Two of the most important machine-dependent features are the word size, i.e. the number of bits in a word, and the machine base. The latter is the base to which machine words are conventionally represented. The machine base is usually octal, and this will be assumed in examples in this manual, but there are many other possibilities, e.g. binary, hexadecimal.

Chapter 2 The UNRAVEL language

Before describing the details of UNRAVEL it is best to show a complete program. Clearly the reader cannot be expected to understand all the details of the program at this stage, but the example does illustrate the general form of the program.

2.1 A sample program

The sample program relates to the example quoted in the previous Chapter concerning the table of uses of I/O devices. It is assumed that the table is pointed at by location 41 (relative to the current base). Some of the important UNRAVEL operations used in the program are indirect addressing (the colon operator), shifting (the ↑ operator) and masking (using the & operator) with constants (octal constants in this example).

```
      LET   TABLE = :41
      REM   LOOP,  X  GOING  FROM  0  TO  29
      LET   X  =  0
10    "DEVICE ";D  X; " "
      LET   STATE =  :(TABLE + X)&060000000
      IF  STATE  =  0  "FREE" ; GOTO 20
      IF  STATE  =  1  "BUSY"
      IF  STATE  =  2  "WAITING"
      LET   PRIORITY = (:(TABLE + X)&017700000)↑-15
      IF    PRIORITY > 10  NL 2; "BEWARE: PRIORITY TOO HIGH";NL 2
      " AT PRIORITY  LEVEL "; D  PRIORITY
      REM ASSUME POINTER TO USER POINTS AT NAME PACKED INTO 2
                                                        WORDS
      LET   USER = :(TABLE + X)&077777
      ".    USER  IS  " ; C :USER; C :(USER + 1)
20    NL
      IF  X  <  29  LET  X = X+1;  GOTO  10
```

The output would consist of thirty lines of a form such as

```
DEVICE  0  FREE

DEVICE  1  BUSY AT PRIORITY LEVEL 3. USER IS CU/RO99

DEVICE  2 .....
```

2.2 Statement format

Statements are terminated with a semicolon or by the end of a line. When a statement is terminated with a semicolon further statements may follow on the same line. Statements may optionally be preceded by a label, which can be any non-negative integer. The rules for spacing are flexible and natural and should not constrain the user. In detail the rules are as follows:

(a) Any number of redundant spaces and/or tabs may be placed between the constituent parts of statements.

(b) In cases where an identifier is immediately followed by another identifier or by a constant it is necessary to place at least one space or tab in between (e.g. after LET in LET A=1).

Note that it is not necessary to place any spaces or tabs before a statement if it is not labelled, but it is best to do so as a program is easier to read if statements are indented to make labels stand out.

2.3 Variables

There is no concept of data type in UNRAVEL. A variable is simply a word of information. The user can choose, as the names of his variables, any identifiers that do not start with the letter Z. Hence the following are acceptable variable names: A, A4, A4PT, TABLEPOINTER. There is no restriction on lengths of names, and names do not need to be declared. (Since they have no data type there is nothing to declare.) Identifiers beginning with Z are reserved for the names of system variables. Each implementation of UNRAVEL will have its own system variables, though two variables, ZGO and ZBASE, are common to all implementations. The purpose of ZGO is to prevent endless loops. ZGO is initialized to some set value, say 10000. When running a program, UNRAVEL maintains a count of the number of backward jumps (including subroutine jumps) it has performed, and if this count ever exceeds the value of ZGO then a message is given and the run aborted. (The count is, in fact, set back to zero every time a statement is encountered that has never previously been executed since it was compiled. This helps prevent false diagnoses of endless loops, though such false diagnoses may occur if a program is re-run without being re-compiled.)

The purpose of ZBASE is to aid indirect addressing. All indirect addresses are taken relative to the current value of ZBASE.

The remaining system variables serve one of three purposes:

(a) To point at useful information. System variables may be set to point at where the current program starts, where its variables are, etc. These may be useful settings for ZBASE.

(b) To control the system, e.g. I/O options.

(c) To preserve information, such as the contents of registers, that is valuable in a dump but might be destroyed by UNRAVEL.

The uses and initial settings of the system variables for each implementation are described in Chapter 5.

There is no syntactic restriction on the use of system variables, though the user must be careful that he understands what he is doing if he changes their values.

The system variables are initialized to appropriate values at the start of each run, and all the remaining variables, i.e. those defined by the user, are set to zero. Hence the user does not need to perform his own initialization for those variables he wishes to start at zero.

2.4 Constants

Unsigned integers may be used as constants. If the integer starts with the digit zero it is evaluated to the machine base; otherwise it is taken as decimal. Thus, for example, on an octal machine 077 would be the same as 63 and on a hexadecimal machine OA9 (where A means ten) would be the same as 169.

(Two very minor points. In the unlikely case of a label starting with a zero it is evaluated to the machine base. On a hexadecimal machine it is advisable to avoid possible ambiguities by leaving a space when a constant is immediately followed by an identifier, e.g. IF X = OABC D Y; .)

2.5 Expressions

Expressions involving constants, variables and operators may be constructed in the normal way. Parentheses may be used freely. The available binary operators, in order of precedence, are

(1) ↑. Logical left shift. The result of the expression E1↑E2 is the value of E1 shifted left E2 binary places (e.g. 5↑3 is 5 x 2^3 is 40). If E2 is negative a right shift is performed (e.g. 40↑-3 is 5). If E2 is larger in magnitude than the word size then the result will always be zero.

(2) comma. The "field" operator, designated by a comma, extracts a field from a word. Since each machine divides words into fields in different ways, the field operator is machine-dependent and is therefore described in Chapter 5. The main purpose of the field operation is to supply a convenient shorthand notation for commonly used masks and shifts.

(3) &. Logical bit by bit "and" operation (e.g. 037 & 071 is 031).

(4) * and /. Multiply and divide.

(5) + and -. Plus and minus.

If two successive binary operators have equal precedence the leftmost is done first (e.g. 4 - 3 - 2 is (4 - 3) - 2).

There are, in addition, two unary operators. The first of these is the colon operator, which performs indirect addressing. The value of the expression :E1 is the contents of the word whose address is given by adding the value of E1 to the value of ZBASE. See Chapter 5 for further details.

The second unary operator is unary minus, which covers cases such as

LET X = -1

Any number of unary operators may be attached to an operand, for example --:-::X. In such cases the order of evaluation is the natural right to left one. Thus :X+1 is taken as (:X)+1. Hence users should be careful to write :(X+1) if they want to address the word at offset one from the pointer X. The following examples show further facets of the precedence rules:

(a) X&Y,Z is X&(Y,Z)

(b) A-B-:C is (A-B)-(:C)

(c) A*-B/C is (A*(-B))/C

Chapter 5 contains further information about operators, for example the effects of overflow or division by zero and the way indirect addresses are taken.

2.6 Input and output

UNRAVEL requires one input stream and two output streams. The input stream supplies the source program. The two output streams are the results stream and the messages stream. The former is used for the results printed out when the program is run and the latter is used for error messages or other informatory messages. In practice the two output streams might not be differentiated; they might, for example, both go to a line-printer.

Chapter 5 gives full details of how the input/output streams are defined.

2.7 Statements

The following is a list of all the allowable statements.

2.8 The null statement

Null statements have no effect on program execution. Their main use is as blank lines to improve program layout. It is also sometimes useful to place a labelled null statement, e.g.

```
999
```

at the end of a program. In this case a

```
GOTO 999
```

would be equivalent to a stop.

2.9 The REM statement

General Form REM characters

Example REM THIS FINDS THE STATUS TABLE

REM statements are used to place comments in a program. They are treated as null statements. The comment cannot involve a semicolon, as this acts as a terminator.

2.10 The LET statement

General Form LET variable = expression

Example LET X = X + 1

The value of the expression is assigned to the variable.

2.11 The GOTO statement

General Form {GOTO / THEN} label

Examples GOTO 123

THEN 16

Jump to the given label. Note that GOTO can have no spaces in it. THEN is an alternative name.

2.12 The GOSUB statement

General Form GOSUB label

Example GOSUB 100

This is a subroutine call. Subroutine calls in UNRAVEL work exactly like those in BASIC. In essence a GOSUB statement works exactly like a GOTO except that a return link is placed on a stack.

2.13 The RETURN statement

General Form RETURN

This unstacks an item from the stack used by the GOSUB statement, and goes to the statement designated by this item, which will be the statement immediately after the last executed GOSUB statement. If the stack is empty, it is an error.

2.14 The PROG statement

General Form PROG character string

Example PROG RIDDLED

The action of this statement is totally machine-dependent. Typically its action is to set certain system variables to point at the locations where the named program (RIDDLED in the above example) resides. Any spaces and/or tabs immediately after PROG are not taken as part of the character string but are ignored. See Chapter 5 for details.

2.15 Introduction to the output statements

The philosophy behind the output statements is that the format of output is totally under the user's control. The output routines do not, therefore, do such things as automatically add extra characters such as spaces and tabs round each number that is printed. The fact that control has been taken away from the output routines and given to the user means that the user sometimes has more writing to do than in some programming languages. He needs, for example, to specify where all the spaces, tabs and newlines are to occur.

All the printing statements use the results stream. If a line becomes too long (e.g. because the user has forgotten to specify any newlines) some implementations will automatically insert a newline so that the rest of the line is not lost; others will simply ignore the rest of the line.

2.16 The string statement

General Form "character string"

Examples "THIS IS X"

"SEMICOLONS ARE ALLOWED;"

This prints the character string within the quotes. The character string may involve semicolons and these are not taken as terminators. It cannot however include any quotes. A null character string is allowed (but is of no obvious use). Spaces, tabs, etc. within the character string are printed exactly as they occur.

2.17 The TAB, NL and QUOTE statements

General Forms

```
TAB
QUOTE
NL
TAB    expression
QUOTE  expression
NL     expression
```

Examples

```
NL
NL     3
TAB    X+6/Y
```

As can be seen, each of these statements can optionally have an expression as its argument. If the argument is omitted one tab, quote or newline is printed. Otherwise the value of the expression gives the number of tabs, quotes or newlines to be printed. If the value is not positive, nothing is printed. Thus

```
TAB    0
```

is a null statement, and

```
TAB    1
```

is equivalent to TAB.

The purpose of the QUOTE statement is to make up for the restriction that quotes are not allowed in string statements.

2.18 The C, D and M statements

General Forms

```
C      expression
D      expression
M      expression
```

Examples

```
C      :(PTR+3)
D      TABPT
M      :(PTR+OFFSET),3
```

The three statements respectively print the value of an expression in character, decimal and machine format (i.e. to the machine base). (Since on, say, an octal machine it would be more natural to use the letter "O" rather than "M" to get an octal print-out, there is a facility for each implementation to have its own synonym for M. See Chapter 5 for details.) In none of the three cases is any extra tabs or spaces printed before or after the value of the expression. When decimal format is used, redundant leading zeros are suppressed and a sign is printed only if the value is negative.

The form of printing for the C and M statements is largely machine-dependent, but one general point can be made. The number of characters or digits is normally fixed (e.g.C might interpret a value as 4 packed characters and M might always print 8 octal digits), but if the last operation that is performed in the expression is the field operation then the printing is usually truncated. This is called <u>limited-field</u> printing, and is best illustrated by an example. Assume that the field operation is defined such that

```
M     X,999
```

means print a certain 2-bit field of X. Then it would obviously be foolish to output 8 octal digits, since only one is needed. Hence the general rule is that on a limited-field C or M statement, printing is limited to sufficient characters or digits to cover the field.

If the user wishes to force limited-field printing, he can, of course, add a field operation to the end of his expression. If he wishes to inhibit it, he can add some redundant operation to the end, e.g.

```
M     X,999+0
```

Note that limited-field printing only applies when the field operation is the last <u>executed</u> operation in the expression. Thus

```
M     2+X,999
```

would <u>not</u> cause limited-field printing since the addition operation, having the lower precedence, is executed after the field operation.

2.19 IF clauses

Any statement can be preceded by one or more IF clauses of form

$$\text{IF} \quad \underline{\text{expression}} \left\{ \begin{array}{c} = \\ > \\ < \\ \text{LE} \\ \text{GE} \\ \text{NE} \end{array} \right\} \underline{\text{expression}}$$

where the relational operators have the obvious meanings. If any of the IF clauses attached to a statement does not hold then control skips to the next <u>line</u>.

The following example illustrates this

```
IF X GE 4   IF X LE 7   "X IS BETWEEN 4 AND 7"; NL

IF TYPE & 8 = 8 "IS A MAN"; IF SALARY > 4000" OF WEALTH";GOTO 100
```

In the first example the string and the newline that follows are printed only if X is greater than or equal to four and less than or equal to 7. In the second example the GOTO 100 is only executed for men of wealth.

Note that UNRAVEL differs from some programming languages in that IF clauses do not have a THEN following them. However the BASIC syntax

```
IF X>Y THEN 100
```

is acceptable since THEN is a synonym for GOTO.

2.20 Sequence of operation

The sequence of operation of UNRAVEL is as follows. Firstly the program supplied on the input device is compiled. Then, without any further command, this program is automatically run. At the start of each run two newlines are sent to the results stream. A run is ended either by a fatal error or by control reaching the end of the program.

Chapter 3 Error messages

There are two types of error: syntax errors, which are detected when the program is being compiled, and run-time errors, which are detected when the program is being run.

3. 1 Syntax errors

Syntax errors cause the current statement to be ignored. (If the statement is labelled and the label is correct then this will not be ignored. Similarly if an IF clause precedes.) Syntax errors do not prevent a program being run. The following is a complete list of error messages that correspond to syntax errors:

(a) UNMATCHED PARENTHESES.

(b) WRONG SYSTEM VARIABLE. A variable name begins with Z and is not the name of a system variable.

(c) MISSING QUOTE. This error, which applies to the string statement, is only detected at the end of a line.

(d) MIS-USE OF LABEL. The same statement has two labels or the same label is used twice.

(e) INCORRECT EXPRESSION.

(f) STATEMENT WRONGLY TERMINATED.

(g) ILLEGAL SYNTAX.

The last three messages are general ones covering a multitude of situations. They do not always indicate the true cause. For example the digits 8 and 9 will act as terminators for an octal constant and might in turn lead to message (f).

3. 2 Run-time errors

Of the run-time errors, some cause the run to be abandoned while others are not so fatal. Errors of the fatal kind are:

(a) SUSPECTED ENDLESS LOOP. The number of backward jumps has exceeded the value of ZGO.

(b) ILLEGAL RETURN. A RETURN statement has been executed when the stack of return links is empty.

(c) STORAGE EXHAUSTED. (This message can also occur at compile-time but, being fatal, is classed as a run-time error.) An endless recursive loop or a program that is too large for the available storage are possible causes of this error.

(d) REFERENCE TO UNDEFINED LABEL number. Labels are checked at the very start of a run, and, if a label appears on a GOTO or GOSUB statement without being defined, this error occurs.

(e) CANNOT RUN. Program has not been sucessfully compiled because of (c) and (d) above and therefore cannot be run.

The following run-time errors, all of which occur during operations within expression evaluation,do not stop the run. The result of the offending operation is assumed to be zero.

(a) DIVISION BY ZERO.

(b) ILLEGAL FIELD CODE. The field operator has an illegal second operand.

(c) ... IS WRONG ADDRESS. Illegal indirection - e.g. out of range.

Chapter 4 Examples

This Chapter shows some short examples that illustrate the usage of the main features of UNRAVEL.

4. 1 Output statements

Output statements usually come in groups, consisting of strings, values, newlines, etc. If a string precedes a value then it is usually best to put a space at the end of the string to separate the two. Similarly if a string follows a value. For example

```
"X HAS VALUE ";D:40;", WHICH POINTS AT ";D::40;NL
```

With a little effort it is possible to achieve quite pleasing output formats. For example assume that the variables LASTBL, STACKPT and TOPPT are stored at offsets 8, 12 and 16 from the current base, respectively. STACKPT and TOPPT point at the start and end of a stack, which contains decimal values. LASTBL points at some intermediate point on the stack. The following program prints out the stack in a diagramatic form such as

```
STACKPT ------>  3
                 7
                 1
LASTBL  ------>  2
                 2
                 1
                 4
TOPPT   ------>  0
```

The program is

```
    LET  X= :12
    "STACKPT ------> "
 10 D:X ; NL
    LET  X= X+1
    IF X= :8   "LASTBL ------>   "; GOTO 10
    IF X NE:16  TAB 2; GOTO   10
    "TOPPT ------>  "; D:X;NL
```

4.2 Subroutines

The following example illustrates the use of a simple subroutine

```
       LET  PARAM =  :20
       GOSUB  100
       LET  PARAM =  :21
       GOSUB  100
       LET  PARAM =  :22
       GOSUB  100
       GOTO   999
       REM  THIS  IS  THE  SUBROUTINE
  100  IF  PARAM = 1   "RED"; NL; RETURN
       IF  PARAM = 2   "GREEN"; NL; RETURN
       "ILLEGAL COLOUR"; NL
       RETURN
  999
```

Chapter 5 UNRAVEL on the ICL 4130

There are two implementations of UNRAVEL for the ICL 4130. One is called KUNRAVEL and works as a KOS sub-system and the other is called NUNRAVEL and runs directly under the NICE executive. Apart from their operating environment the two implementations are almost identical not only in the machine-independent features but also in the machine-dependent features described in this Chapter. Hence except where otherwise stated all facilities described here apply to both KUNRAVEL and NUNRAVEL.

5. 1 Machine-dependent operations

The word size for the ICL 4130 is 24 and the machine base is octal. Arithmetic operations work as described in the previous Chapters. There is no special facility for dealing with floating-point representations.

The initial values of ZBASE and all the other system variables that are possible settings for ZBASE are absolute addresses containing bit 21. For the indirection operator the address to be accessed (i.e. the result of adding the operand to ZBASE) is checked to see that it includes bit 21 and that the address part is less than the contents of absolute location 165 (STORESIZE). An error is forced if either of these conditions does not hold. If the user wishes to create his own settings for ZBASE it is therefore best to make these absolute addresses with bit 21 in them. In particular

```
LET ZBASE = 04000000
```

could be used to look at DES-2.

The field operation on the 4130 is defined such that E1, E2 means the last E2 bits of E1 if E2 is positive and the first -E2 if E2 is negative. Thus for example

```
012345671,8 is 0271
012345671,-8 is 051
```

If the absolute value of E2 exceeds 23 or is zero an error is forced.

5. 2 Printing formats

"O" (the letter "Oh" not the digit zero) is allowed as a synonym for the "M" statement.

In normal character printing the value of the operand is interpreted as four six-bit characters. All characters are taken as in-shift. If a shift character is encountered the character "↑" is printed in its place. In limited-field character printing, the value of the operand is always interpreted as a single 7-bit character, irrespective of what field code is used. Hence the last seven bits of the operand are extracted and if the first of these bits is one the character is taken to be in the out-shift set. (The character "↑" is printed in place of the shift characters given by octal codes 76, 77, 176 and 177). Hence, for example, if the value of the variable W is to be interpreted as a 7-bit character the most natural way to do this is to write

```
C W,7
```

Limited-field octal printing works exactly as described earlier in the manual, i.e. enough octal digits to completely cover the field are printed.

If any lines of output become too long an extra newline is automatically inserted.

5. 3 Operating instructions for KUNRAVEL

Operating instructions for KUNRAVEL and NUNRAVEL are, of course, completely different.

KUNRAVEL runs as a KOS sub-system and is entered by the KOS command

&ENTER KUNRAVEL <u>number</u> <u>DR-spec</u>

where <u>number</u> gives the KOS device number of the default input device of the KOS sub-slave to be examined. The <u>number</u> may be omitted, in which case the current sub-slave is assumed. If a specified <u>number</u> does not correspond to the default input device of an existing KOS sub-slave then KUNRAVEL returns to command status, giving the error message NO JSB. The significance of <u>number</u> is that it determines the initial settings of ZBASE and ZJSB (q.v.). Many KUNRAVEL programs, however, look at executive-level information rather than sub-slaves and these programs start by resetting ZBASE. In such cases the initial setting of ZBASE is immaterial, and number can sensibly be omitted.

At the end of a compilation a run is commenced automatically, even if errors have occurred. Use of I/O is according to the usual KOS conventions. A break at any time causes a return to command status within KUNRAVEL. There are two subsidiary commands, namely

&SCR   number   DR-spec

which restarts from scratch (i.e. it is identical in effect to ENTER KUNRAVEL but saves the overheads of reloading), and

&RUN   number   DR-spec

which re-runs the program previously compiled. In each case the argument list has the same meaning as for the ENTER KUNRAVEL command (except that RUN needs no data device). If compilation has not been completed then RUN commands are rejected.

It is not possible to change or extend a previously compiled program.

On entry KUNRAVEL borrows the largest available block of user's workspace and uses this for the compiled program, etc. It is therefore not very useful to use KUNRAVEL to examine the KOS sub-slave in which it is running as it will end up by looking at itself.

5. 4 Operating instructions for NUNRAVEL

NUNRAVEL runs under NICE and is best used under BATCH, though it is possible to run it directly from the control teleprinter. It uses ACIO for its input and its results output. Channel 36, which defaults to cards, is used for input and channel 34, which defaults to the line-printer, is used for output. Input that consists of several different parts can be dealt with by reassigning channel 36 during the input (in a similar way to the use of channel 1 for NEAT), for example

```
&NUNRAVEL;
LET  PARAM=32
&ASSIGN;36;DC;2;UNRAV1,XXX;
&ASSIGN;36;DC;2;UNRAV2,XXX;
"SUCCESS";NL
↑↑↑
```

Note that the card terminator (↑↑↑) must always be present if input is initially from cards.

Error messages go straight to the line-printer.

When it commences execution, NUNRAVEL increases LOWADD by 2000 and uses the area thus reserved for its workspace. This default allocation can be overridden by a numerical parameter on the call of NUNRAVEL, e.g.

&UNURAVEL; 4000;

might be used for a very long program.

Other possible parameters to the call are

L meaning list (on the lineprinter) all the input.

LC meaning list that part of the input that comes from cards.

SET meaning stop at the end of compilation (see later).

Ordering of the parameters is immaterial, e.g.

&NUNRAVEL;SET,1000,LC;

is allowed. Illegal parameters are ignored.

When using NUNRAVEL to examine core after a certain occurrence there are two possible sequences of operation

(a) Occurrence happens; NUNRAVEL is loaded; compiles program; runs.

(b) NUNRAVEL is loaded; compiles program; occurrence happens; NUNRAVEL runs.

Sequence (b) is by far the better since the act of loading NUNRAVEL overwrites a lot of core, and this might be just the core that needs to be looked at. Thus the SET parameter is provided. This causes NUNRAVEL to surrender control at the end of compilation, but to remain poised for a later run. The next entry to NUNRAVEL is then taken as a command to run the program. Thus a typical sequence of BATCH cards when NUNRAVEL was examining the effects of running a program called FUNNY might be:

```
&NUNRAVEL;SET;
UNRAVEL program
↑↑↑
&FUNNY;
&NUNRAVEL;
```

Once NUNRAVEL has been SET all subsequent calls are taken as instructions to re-run the same UNRAVEL program. If a new program is required NUNRAVEL should be CANCELled and re-loaded. (It is also best to do this even if it has not been SET. Otherwise it will keep taking new workspace areas.)

NUNRAVEL can also be used to supplement PM after a KOS logical error. It should be SET before KOS is entered. When a logical error occurs in a batch run, KOS checks to see if NUNRAVEL is in core and, if it is, enters it with the parameter "KOS". When NUNRAVEL is thus entered it is said to be in KOS-mode. When a KOS-mode run ends, NUNRAVEL returns control to KOS, which then runs PM. It is planned to provide a library of standard NUNRAVEL programs that are useful in these circumstances. If NUNRAVEL is entered with the parameter "KOS" without having been SET, then the error message NOT SET is given.

5. 5 System variables

A list of the available system variables and their uses appears below. Users are advised not to change the values of any of these apart from ZBASE and perhaps ZGO.

ZGO	Limit on backward GOTOs. Initialized to 10000.
ZBASE	Base for indirect addressing. Initialized to value of ZSLAVE (see below) except that NUNRAVEL, when not in KOS-mode, initializes it to the value of ZNICE.
ZNICE	Base of world in which NICE, KOSEX, MCP, etc. operate.
ZSLAVE	Base of KOS sub-slave of interest.
ZJSB	Base of JSB (Job Status Block) of KOS sub-slave of interest.
ZPROG, ZENDPROG, ZSIZEMC, ZMC	describe the current program. Initially zero but re-set by PROG command (q.v.).

When NUNRAVEL runs in KOS-mode, ZSLAVE and ZJSB refer to the KOS sub-slave that caused the logical error. When NUNRAVEL is not in KOS-mode they are irrelevant and are given the initial value zero.

5. 6 The PROG command

The purpose of the PROG command is to set some system variables to point at a given program so that the user can examine it more easily. The user specifies the program name as an argument to the PROG command e.g.

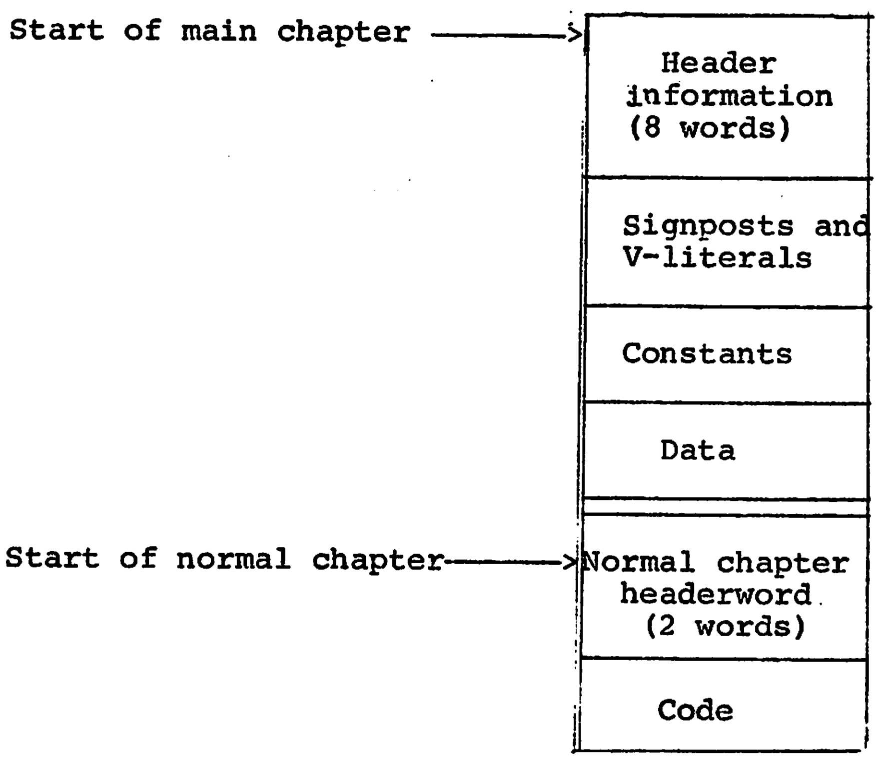
PROG KOSEX

The program name should consist of an identifier of at most eight characters. Any characters beyond this are ignored.

The PROG command works in a similar manner for KUNRAVEL and NUNRAVEL. However NUNRAVEL is slightly simpler and will be described first.

5. 7 PROG for NICE programs

For NUNRAVEL when not in KOS-mode the action is as follows. The program name is looked up in the NICE table to find where in core it lies. The layout of a single-chapter program in core is as follows (assuming it has been assembled by NEATER):



The PROG command causes four system variables to be set to describe this in the following way.

ZMC — is the base of the start of the main chapter.

ZSIZEMC — is the size of the main chapter.

ZPROG — gives the S-value of the start of the normal chapter (i.e. 2 words before the first instruction).

ZENDPROG — gives the S-value of the last instruction in the program. (If the program has a self cancelling interlude this may have been overwritten.)

S-values are as they appear in the world that NICE operates in. Bits 24 to 17 are zero. Note that all chapters are assembled to contain an even number of words, an extra word with value zero being added on to accomplish this where necessary. Hence ZENDPROG may refer to an instruction that is one beyond what the programmer thinks to be his last instruction.

It is often useful to print out all the variables in a NICE-based program, but it can be seen from the above picture that the position of the variables depends on the number of constants and V-literals. The relative position of data is given, in octal, at the end of a NEATER listing. However if a program is continually being changed this is not very useful. A much better way of finding data is to place a unique value as the last constant, e.g. 0:76543210, and search for this. Assuming this has been done, the following UNRAVEL program would print out, in decimal, the values of all the variables in a program. Each value is preceded by its octal offset within the data area so that it could be compared with a NEATER listing.

```
   PROG  XXX
   LET   ZBASE = ZMC
   REM   START SEARCHING FROM END OF HEADER INFORMATION
   LET   X = 8
10 IF  :X NE  076543210  LET X = X+1; GOTO 10
   LET   X = X+1
   LET   VARNO = 0
20 M VARNO ; TAB; D :X; NL
   LET X = X+1
   LET  VARNO = VARNO+1
   IF X<ZSIZEMC GOTO 20
```

The usage of ZPROG and ZENDPROG is for interpreting subroutine links. Assume, for example, that X points at a storage location that contains a link to a point in the program ZZZ; then the following UNRAVEL program would give the octal offset of the link in the program. This could then be compared directly with a NEATER listing.

```
    PROG  ZZZ
    LET  LINK = :X& 0177777
    IF LINK>ZPROG IF LINK LE ZENDPROG GOTO 100
    "LINK IS OUTSIDE PROGRAM ZZZ. VALUE= ";M LINK: GOTO 999
100 "LINK IS AT ";M (LINK-ZPROG)/2; "N ON A NEATER LISTING"
999 NL
```

Note that ZPROG and ZENDPROG are S-values and cannot be used as bases. However ZMC+ZSIZEMC is the base corresponding to ZPROG and ZMC+ZSIZEMC+(ZENDPROG-ZPROG)/2 corresponds to ZENDPROG.

5. 8 PROG for KOS programs

NUNRAVEL when in KOS-mode and KUNRAVEL look at KOS programs. The program name is therefore searched for on the KOS table, not the NICE table. Otherwise the workings are exactly as described previously. ZPROG and ZENDPROG contain bit 17 if KOS is in common program mode. When KOS is not in common program mode these S-values are set as they would appear to the specified KOS sub-slave, and therefore not as they would appear to NICE.

Two extra facilities are available. Firstly,

PROG

on its own means the currently used sub-system or, if none exists, then COMMAN. (This information is derived using the fixed location USUBSYS at offset 180 in the sub-slave.)

The other extra facility, which is only available in NUNRAVEL in KOS-mode, is that

PROG LOG

means the program that was responsible for the logical error that caused NUNRAVEL to be entered. (This information is derived by looking at the value of the S-register when the logical error occurred. If this S-value is outside any existing sub-system the error action described below is taken.)

In the case of both the extra facilities a message is printed to tell the user which program has been selected, e.g.

PROG SET TO KOSML1 IN LINE 13

There is one extra restriction. If KOS is running under DES-1, PROG commands are forbidden. If they are used they give rise to the error message described in the next Section.

The user should be sure that, when using PROG under KUNRAVEL, the program to be looked at will remain in core. If it is released and overwrittem while KUNRAVEL is looking at it, the results will be disastrous.

5. 9 PROG errors

In all cases if the program required by PROG is not found then the error message

NO PROGRAM FOUND IN LINE ...

is output. The run continues and none of the values of system variables is changed.