THE ML/I MACRO PROCESSOR

Implementing software using the LOWL language

Supplement 2 UNRAVEL

P.J. Brown
Computing Laboratory
University of Kent at Canterbury
September 1972.

Introduction

UNRAVEL is a programable dumping program. It has proved useful in two main areas:

(a) providing a better debugging aid than a traditional dump in binary (octal, hexadecimal, etc.) notation.

(b) monitoring dynamic systems.

Anyone wishing to implement UNRAVEL must first read carefully the User's Manual of an existing implementation, with particular attention to the machine-dependent and operating system dependent features that are described in Chapter 5 of each User's Manual.

On reading such a manual it will be apparent that UNRAVEL contains a reasonable degree of machine-dependence. (There may even be a few machines, notably those where the concept of a "word" is not meaningful, for which the basic design of UNRAVEL, as mirrored by its MI-logic, is unsuitable as it stands, and an implementation via LOWL for these machines would not be useful.)

In the machine-independent implementation of UNRAVEL described here, all the machine-dependent features have, of course, been placed under the control of the implementor. This has meant that the MD-logic is comparatively large. It is, however, still much smaller than the MI-logic, and so the effort of mapping the MI-logic is generally worth while.

Selection of system variables

One of the first tasks of the implementor is the selection of the system variables that will be available on his implementation.

As a guide, the following are some possible uses:

(a) Preservation. The action of UNRAVEL may destroy some of the things it is supposed to be dumping, particularly the contents of registers. These can be preserved when UNRAVEL is entered and made available to the user as system variables, e.g. ZACC1 might be accumulator one.

(b) Pointers. System variables might be set to point at important areas of store that a program might wish to examine, e.g. system tables, buffers, workspace, stacks, etc.

(c) Control. The variable ZGO provides run-time control of loops. The user might wish to provide his own variables of a similar nature, e.g. control of output (line length, device to be used, etc.).

Extensions to LOWL

The following are the extensions to LOWL required by the MI-logic of UNRAVEL.

1. Character set

The character set of UNRAVEL uses the additional character "↑" (shift operator), which occurs as operand to the CCL statement.

2. Character constant

UNRAVEL needs an additional named character constant called BASREP. This is concerned with the M statement in UNRAVEL, which means print in machine format (e.g. octal or hexadecimal). It is convenient to have a machine-dependent synonym for M, e.g. O (for octal) or X(for hexadecimal), and BASREP provides for this. The value of BASREP must be the internal code for a letter; if no alternative to M is required then the value of BASREP should be the internal code for M.

3. Subsidiary macro of the OF macro

The macro LMBS, which is, like LNM and LCH, a subsidiary macro of the OF macro, should be assigned the value of the machine base. For example on an octal machine LMBS should be defined as eight.

4. The ZNAMES statement

The MI-logic provides only for the names of those system variables which should be present in all implementations. These are ZGO and ZBASE. The replacement for the ZNAMES statement should supply the machine-dependent ones. The form of these should be the name followed by its length, these name-length pairs being placed end to end. For example if the names (which must be identifiers beginning with Z) are ZTABLE, Z, and ZBUFF then ZNAMES might be replaced by

```
STR     'ZTABLE'

CON     OF(6*LCH)

STR     'Z'

CON     OF(LCH)

STR     'ZBUFF'

CON     OF(5*LCH)
```

ZNAMES has no argument.

5. The MULT and DIV statements

The MULT and DIV statements have the forms

```
MULT    V    Multiply  A  by value of  V.

DIV     V    Divide  A  by value of  V.
```

Overflow can be ignored. The MI-logic detects division by zero as an error and does not call DIV in this case.

Introduction to the MD-logic

The I/O requirements of UNRAVEL are very simple. One input stream is needed, and there are two notional output streams, the messages stream and the results stream, as described in the LOWL manual. For initialisation UNRAVEL needs only the common initialisation code described in the LOWL manual. The stack area is mainly used for storing the source program, which is compiled into reverse Polish form. A reasonable default size for this is two thousand words, though the user should have a facility for asking for more.

The implementor may wish to introduce his own error or warning messages into the MD-logic, for example on detecting an I/O error. The MI-logic maintains a variable called LINECT which gives the line number of the program statement currently being compiled or run, and it may be useful to include this in the implementor's own messages.

The MD-logic subroutines are listed below.

The MDQUIT subroutine

The MDQUIT subroutine is entered when the run of the program has ended (either normally or because of an error). Its action should be to close all I/O - there may be an incomplete line on the results stream waiting for output - and release resources. After doing this it should not return to the point of call but should surrender control to the containing system, or, in an interactive environment, perhaps communicate with the user to ask him what to do next.

It is possible to re-run a program without re-compiling it. This is done by branching to the label RERUN in the MI-logic. Hence an implementation might provide a command of form

RERUN <u>param</u>

where <u>param</u> changes the run in some way, e.g. changes the area of core being examined, or the results stream.

The MDRIN subroutine

The MDRIN subroutine, which has no parameter and only one exit, is used to perform run-time initialisation. It is called at the stage the program has been compiled and is just about to be run. The main action of MDRIN is to set initial values for all the system variables. These initial values should be stacked on the backwards stack, the stacking being done in the opposite order to which the variables are named. (The names are those generated by the ZNAMES statement, followed by ZBASE and ZGO.) For example, if the ZNAMES statement is defined in the way shown earlier, the initialisation might be performed thus

```
LAL       10000

BSTACK              initial value for ZGO

LAL       0

BSTACK              initial value for ZBASE

LAV       ...

BSTACK              initial value for ZBUFF

...
```

BSTACK	initial value for Z
...	
BSTACK	initial value for ZTABLE

In some operating environments MDRIN may be given some other actions as well. For example, it is possible to delay selection of the results stream until this stage since the only output produced during compilation is error messages. Moreover it is possible to close the input stream since no input is needed at run-time.

In one implementation, a "SET" mode has been made available; this allows a program to be pre-compiled ready for a later run. In this case MDRIN surrenders control to the containing operating system. The MI-logic is subsequently re-entered at the label RERUN each time a run is required. Of course, the MI-logic and its variables and stacks have to remain unchanged during such breaks. (Note that when a run is started by jumping to the label RERUN, MDRIN is subsequently called again. The MD-logic should therefore set a switch to tell MDRIN whether a SET or a run is in progress.)

During the run the variable SYSPT always points at the place on the backwards stack where the value of the first system variable, ZGO, is stored. The value of the Nth system variable is therefore at the address given by

SYSPT - OF((N-1)*LNM)

This might be used to look at any system variables that are used for control purposes, e.g. the fifth variable might give the current device number for results output.

The MDLINE subroutine

MDLINE is a subroutine to read lines of input, i.e. the UNRAVEL program to be compiled. It has two exits and uses exit 1 if input is exhausted. Otherwise it reads a line of input into a buffer, sets BUFFPT to point at the first character of the buffer and uses exit 2. (BUFFPT should be reset on each call of MDLINE, since the MI-logic changes it.) MDLINE is responsible for reserving space for the buffer. The text in the buffer should be terminated with the newline character, as represented by NLREP. Thus, for example, a null input line would cause the buffer to contain simply a single newline character.

If an implementation supports such options as listing of the input or switching of input streams (e.g. taking part of the input from disc), then MDLINE is responsible for these.

The MDERCH subroutine

The MDERCH subroutine, which has one exit, outputs on the messages stream the character in C.

The MDERNM subroutine

The MDERNM subroutine, which has one exit, outputs the number in A on the messages stream. This number should be converted to a decimal representation. It may be negative, in which case a minus sign should be output in front of it. Redundant leading zeros should be suppressed.

For example the MI-logic code

```
MESS     'ABC'

LCN      NLREP

GOSUB    MDERCH,X

LAL      0

SAL      123

GOSUB    MDERNM,X

MESS     'XYZ$'
```

should generate the two lines

```
ABC

-123XYZ
```

The MDOUCH and MDDOU subroutine

The MDOUCH and MDDOU subroutines are identical to the MDERCH and MDERNM subroutines, respectively, except that they use the results stream rather than the messages stream. In the case of MDOUCH the character to be output may be a tab.

The MDMOU subroutine

The MDMOU subroutine, which has one exit, outputs on the results stream the "machine" representation of the word in A. This representation might be, for example, six octal digits if the object machine base is octal and the word size 18 bits.

MDMOU should test if printing is to be limited-field (see User's Manual) by looking at the variable TEMP. If TEMP is not zero limited-field printing applies, and the variable TYPE, which gives the value of the second operand of the most recent field operation, should be examined in determining how many digits to print. (Alternatively it might be easier for the implementor to make the MDFLD subroutine (q.v.) set some variable that MDMOU can examine to give a more direct indication of the number of digits to be printed.)

The MDCOU subroutine

The MDCOU subroutine, which has one exit, outputs on the results device a character representation of the word in A. Since most machines allow several characters to be packed into a word, MDCOU will normally output a string of characters. This output of characters should be restricted by limited-field printing in the same way as in MDMOU, i.e. the variables TEMP and TYPE should be examined. Thus if the field code 6 selects one of the characters packed into a word then

```
C X,6
```

should output only one character.

The MDNUM subroutine

The MDNUM subroutine has two exits and a parameter in A. The parameter is either ten or the machine base as given by LMBS. MDNUM tests the character pointed at by BUFFPT. If it is a digit less than the parameter it puts the equivalent decimal value in TEMP and uses exit 2; otherwise it uses exit 1.

In the case where digits have contiguous internal codes, MDNUM can be encoded thus

```
SUBR    MDNUM, PARNM,2

LCI     BUFFPT,X

Subtract from C the internal code of the
digit zero and place the result in A
```

```
        STV     TEMP,P
        CAL      0
        GOLT    MDEX1,4,X,X              q
        LAV     TEMP,R
        CAV     PARNM,X
        GOGE    MDEX1,1,X,X
        EXIT    2,MDNUM
[MDEX1] EXIT    1,MDNUM
```

Note that on a hexadecimal machine where the letters A to F represent the numbers 10 to 15, these letters would count as digits.

The MDSHIF subroutine

The MDSHIF subroutine has a parameter in A and one exit. It should take the word pointed at by LFPT and shift it by the number of bits given by the parameter - positive means left shift, negative means right shift - and exit with the result in A.

Shifts should be logical ones, not arithmetic ones, and should not be circular. Shifts longer than one word size of the object machine should therefore produce an answer of zero.

The MDFLD subroutine

The MDFLD subroutine has a parameter in A and two exits. It effects the "field" operation. The way this is done is chosen by the implementor, but it should be such that any commonly used field can be accessed. MDFLD should take the word pointed at by LFPT, extract the field designated by the parameter, shift the result right so that the rightmost bit of the field is the rightmost bit of the result, and use exit 2 with the result left in A.

If the field code is illegal, exit 1 can be used. The MI-logic will then output the message

ERROR IN LINE ... - - ILLEGAL FIELD CODE -- RESULT ASSUMED TO BE O

and continue the run.

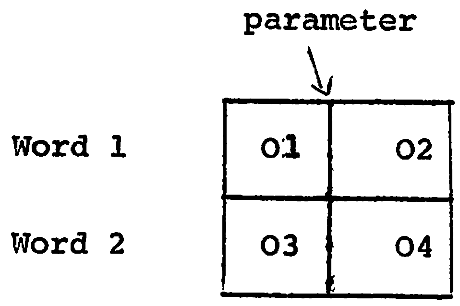
The MDAT subroutine

The MDAT subroutine, which performs indirect addressing, has a parameter in A and two exits. The parameter is the operand of the indirection operator with ZBASE added to it. If this is an invalid address, exit one should be used. The MI-logic will then output the message

```
ERROR IN LINE ... -- ... IS WRONG ADDRESS -- RESULT ASSUMED TO BE 0
```

and continue the run. Otherwise the word pointed at by the parameter should be placed in A and exit two used.

There are many machines where the minimum addressible unit is smaller than a word, for example byte machines. For these machines, the parameter of MDAT might not point at a word boundary. In this case it is suggested that enough bits be extracted from the store to make up a word, making the assumption that a word starts where the parameter points. For example in the situation shown in the following picture



the result should be 0203. (There is a slight problem if Word 1 is the last word in store; in this case it should be assumed that Word 2 contains all zeros.)

The MDPROG subroutine

The MDPROG subroutine, which has a parameter in A and one exit, is used to effect the PROG statement. It should find where the named program is located in store and set appropriate system variables to describe it. (Under operating systems where this is impossible, the PROG statement would need to be given some other meaning. In the worst case the MDPROG subroutine would have to be null, thus making PROG statements equivalent to comments.)

The parameter points at the argument to the PROG command, as supplied by the programmer. This is terminated by the character NLREP. Hence if the programmer writes

```
                    PROG    AB/CD
```

this is compiled into the equivalent of

```
             [XXX]    STR    'AB/CD'
                      NCH    NLREP
```

and at run-time MDPROG is called with a pointer to the character string labelled by XXX. Note that this string is an exact copy of the string supplied as argument by the programmer; the only omissions are spaces and tabs at the very start.

Actually there is nothing to compel MDPROG to interpret the string as a program name. It can be interpreted in any way that seems suitable, thus making MDPROG a generalised machine-dependent statement. Even if the string is generally interpreted as a program name, some exceptions can be made; for example a null string may mean that some standard options should be set. In whatever way the string is interpreted, there are bound to be cases that should be treated as errors. In these circumstances MDPROG is responsible for giving the appropriate error message. It should exit in the normal way.

Documentation

When an implementation is working, the implementor needs to provide documentation for the user. The UNRAVEL User's Manual is written in such a way that all machine-dependent features are described in a single Chapter, Chapter 5. Hence it is only necessary to rewrite this one Chapter (and perhaps the title page and table of contents) for each new implementation.

Final cautionary words

It is very important that a dumping program be robust; if it fails, all is lost. Hence it is desirable that, whatever goes wrong, it should be impossible for UNRAVEL to cause a hardware trap. Hence great care should be taken in coding the routines described earlier, particularly the MDAT subroutine, which might cause an addressing fault. Another possible source of danger is overflow due to the generation of very large numbers.

In one sense security clashes with utility. Clearly a dumping program should itself be as small as possible. On the other hand, it should, as far as possible, be self-contained since routines outside it might be corrupt. It should therefore contain its own I/O routines. The implementor must judge how to balance these conflicting requirements for his particular circumstances.