TRIUMF	**UNIVERSITY OF ALBERTA EDMONTON ALBERTA**		
	Date 3-March-1991	*File No* TRI-DNA-91-3	
Author P. W. Green			*Page* 1
Subject The NOVA Operation Sequence (OPSEQ)			

The NOVA Operation Sequence

The operation sequence (OPSEQ) is the "program" which is executed to analyse an event. Although you can think of it like a FORTRAN program, there is a fundamental difference. The OPSEQ does *not* contain statements to evaluate variables. Variables are calculated *automatically by the system as they are needed* (using the formulae entered in the definition section discussed previously). The OPSEQ deals primarily with:

1. Which histograms are to be incremented.
2. (Optionally) a "Logic Tree" controlling under what circumstances a histogram will be incremented.

OPSEQ Statements

Labels:

Any line in the OPSEQ may be labelled. A label is a unique identifier of from 1 to 12 alphanumeric characters (including **$** and _) terminated by a colon (:). The first character must be alphabetic or $. They are used primarily as the targets of GO TO statements, although they may appear anywhere.

CONTINUE

As in FORTRAN, this is a "do nothing" statement, and is included primarily to provide a place for a label if none is convenient.

ELSE

The ELSE keyword begins the block of statements to be executed if the expression in an IF...THEN statement is FALSE. See the discussion of the Block structured IF statement below.

ENDIF

The ENDIF statement terminates a block structured IF...THEN statement (see below).

EVAL var

This statement forces the evaluation of a particular variable (or group of variables). It is not normally needed, as variables are automatically evaluated by the system as they are needed. It is included for those cases where variables (primarily *conditions*) should be evaluated for every event whether they are "needed" or not (for example, because the values of the software scalers are important). Note that variables are only calculated once (at most) for each event. Thus a complex OPSEQ might end with the statement:

EVAL $ALLCONDS

to ensure that all conditions are evaluated for this event. Those that had been done already during normal OPSEQ execution would not be affected - any that had not been done would be evaluated (and their software scalers updated).

Variables can also be defined (i.e. calculated) by calling a user function (which could have all sorts of other effects, such as incrementing histograms through user subroutine calls). This statement could then be used to force a particular User function to be executed.

GOTO label

As in FORTRAN, this implies that the next statement to be executed is the one labelled *label.* GOTO may be one word or two (GO TO). Note that in this context (and also in the IF statement below) *label* is entered without the terminating colon.

IF (expression) {GOTO} label
IF (expression) THEN ... {ELSE ...} ENDIF

One of the more common ways of conditionally incrementing a spectrum is by skipping over part of the OPSEQ depending on whether a particular logical expression (or variable) is TRUE or FALSE. Two forms of the IF statement are provided for this - IF...GOTO and the block structured IF...THEN.

1. IF...GOTO

The statement following the IF statement is executed if *expression* is FALSE, and control is transferred to the statement labelled *label* if *expression* is TRUE. *Expression* may be either a single variable (of any type) or any arithmetic or logical FORTRAN expression. If it is a variable, it is considered FALSE only if its value is 0 - *any non-zero value is regarded as TRUE.*

Expression may also be a *condition group.* In this case, conditions in the group are tested until one of them is FALSE, in which case the program "falls through" the IF statement to the next statement. Only if all of the conditions in the group are TRUE will the GO TO be executed (i.e. for a group, *expression* is effectively the Logical AND of all of the conditions, and *all of them must be TRUE for the GO TO to be executed.*

The keyword GOTO (or GO TO) is optional - you can use any of the forms:

IF (expression) GOTO label
IF (expression) GO TO label
IF (expression) label

(It should be obvious, at least after reading the next section, that you should not have a label called 'THEN').

There is currently <u>*no restriction*</u> *on branching backwards, so it is possible to get yourself into a loop from which you will never escape. Be careful!*

2. Block structured IF…THEN

The general form is:

```
IF (expression) THEN
        block1
ELSE
        block2
ENDIF
```

If *expression* is TRUE, the group of statements *block1* will be executed, while if it is FALSE, the group of statements *block2* will be executed. In either case, the next statement will be the one following ENDIF. The ELSE part is optional, but every IF…THEN must be terminated by its own ENDIF statement. Block structured IF…THEN statements may be nested to any level.

Expression may be a single variable, logical expression or condition group, as discussed above for the IF … GO TO statement.

NOVA does *NOT* support the ELSE IF statement.

When you are in OPSEQ insert mode, the system automatically indents lines to the appropriate nesting level, and also shows you the nesting level when you LIST/OPSEQ. Two special characters are important.

1. If the first character listed on the line is '>', it means that the nesting level is greater than 20 (this is not an error - it just means that the system would have to space things over too far on the screen).
2. If the first character is '*', it means that the nesting level is *negative* (i.e. you have more ENDIF's than IF's. This *is* an error condition.

Whenever you make changes to the OPSEQ, the system checks that the IF…THEN's, ELSE's and ENDIF's match up OK. If they don't (e.g. an IF with no ENDIF or an ELSE without a corresponding IF), it issues an error message and declares the OPSEQ *non-executable*. You must fix the problem before the system will let you begin analyzing data.

Every IF statement (of either breed) in the OPSEQ is essentially a *condition* in that there is associated with it a unique *software scaler* which records how often the expression was evaluated (i.e. how often you got to the IF statement) and how often it was TRUE. These can be viewed at any time with the LIST/IF or LIST/OPS/FULL command.

INCR

The INCR statement specifies that a histogram (or a *group* of histograms) is to be incremented. Various forms of the statement are supported:

INCR specname BY weight	This specifies that spectrum *specname* is to be incremented. The appropriate channel of *specname* will have the (floating point) value *weight* added to it.

INCR specname weight The keyword *BY* is optional.

INCR specname If *weight* is omitted, the default weight of 1.0 is used.

specname BY weight
specname weight
specname The keyword *INCR* is also optional. If the first word (except for a label) on a line is not a recognized keyword, it is assumed to be the name of a spectrum to be incremented. Note that this implies that keywords *or any unique abbreviations of keywords - such as IN (short for INCR)* should not be used as the names of spectra (you can if you really want to - it just means that you *MUST* use INCR to increment it).

Note also that the *weight* may be either a *variable* or a *constant* and is a *REAL* number. One possible application of using a variable as a weight would be to calculate an on-line analyzing power, incrementing a spectrum by 1.0 for spin up and -1.0 for spin down.

OUTPUT

Output a variable (or part of a vector) to an output "event" which will be written to the file /device specified in the OPEN/OUTPUT command. this provides a means for NOVA to "skim" data (keeping only "good" events) and, optionally, to add additional information to the output event.

The general form of this command is

OUTPUT variable length

Length is redundant for simple (scalar) variables. For vectors, if length is not specified, the entire vector will be output.

If variable is not given, the input data ($RAW) is assumed. Thus,

OUTPUT	Outputs the entire $RAW input vector
OUTPUT X	Outputs the single variable X or the entire vector X (if it is dimensioned).
OUTPUT X 6	Outputs the first 6 elements of the vector X
OUTPUT 12	Outputs the first 12 elements of the input vector $RAW.

Several OUTPUT statements may appear in the OPSEQ. The values are concatenated into a single "event" which is written in standard VDACS format (i.e. the file can be read by NOVA using the IMODE VDACS format). The only restriction is that the total length of the output event must be smaller than 8182 bytes (one VDACS tape block minus the 5 word header).

Editing the OPSEQ

Changes to the OPSEQ may be made by either:

1. Using the VAX editor EDT (from *within NOVA*), using the EDIT/OPSEQ command. This is undoubtedly the preferred option for most users.
2. To allow compatibility with earlier versions (especially command files), and to allow NOVA to be transported to other (non-VAX) environments (if / when I ever get around to that), you can also "edit" the OPSEQ by line number.

Full Screen (VAX-EDT) Editor

The command EDIT/OPSEQ invokes the VAX editor (EDIT/EDT) to edit the OPSEQ. It looks just like EDT (in fact, it IS EDT), *with the exception that there are no disk files involved (everything is done in memory), and in particular, there is no JOUrnal file).* One (very) un-nice side effect of this is that the log file produced by NOVA (NOVAINPUT.LOG) no longer contains a complete record of everything that you did during the session - anything that happened while you were in the EDT editor is lost! I have some ideas of how to deal with this problem, but they are not implemented yet, so BEWARE!

The screen you are shown in the editor will not contain the first line ($BEG_OPSEQ: CONTINUE) of the last line ($END_OPSEQ: CONTINUE) of the OPSEQ. For various technical reasons, these lines must not be deleted from the OPSEQ, so rather than check to make sure you didn't, I just don't give you the option.

If you have your favourite EDT command file which customizes EDT, you must make the (VAX) logical name assignment:

```
$DEFINE EDTINI yourcommandfile
```

before you enter NOVA.

As under VMS, use the (editor) command *EXIT* to leave the editor and save all changes (i.e. what you did in the editor will become the new OPSEQ), and use the command *QUIT* to abort (leave the editor and keep the old OPSEQ).

Single Line OPSEQ "Editing"

To insert a single line into the OPSEQ, just enter the line number followed by the command. For example

12 INCR SX

would *insert* this as line # 12 in the OPSEQ (if line 12 exists already, it is "pushed down", not overwritten).

An input line containing only a line number causes NOVA to enter *OPSEQ insert mode,* and several lines may be inserted. NOVA lists the immediately preceding line, and then accepts OPSEQ lines until a CTRL-Z is entered (or the EOF, EXIT of QUIT command, or a blank line). For example,:

12
INCR SX

```
INCR SY
IF (X>4 & Y < 10) L1
INCR SZ
L1: CONTINUE
EOF
```

would insert lines 12 - 16 in the OPSEQ (higher numbered lines are pushed down - what used to be line 13 is now line 17).

The "empty" OPSEQ contains the two lines:

$BEG_OPSEQ: CONTINUE	(always the first line)
$END_OPSEQ: CONTINUE	(always the last line)

Lines can be deleted from the OPSEQ with the command

DELETE/OPSEQ n1	Deletes only line number n1
DELETE/OPSEQ n1 n2	Deletes lines n1 to n2 inclusive
DELETE/OPSEQ ALL	Deletes the entire OPSEQ
DELETE/OPSEQ	Deletes the entire OPSEQ, but requests confirmation first.

See the NOVA Commands manual for a more detailed description.

Examples

Analysis / acquisition systems are very personal things (if you like the one that you like, and you probably do, you almost by definition abhor the alternative and wild horses couldn't make you change). Therefore, NOVA tries to allow you to tailor your analysis code to (at least resemble) your favourite system. In addition, it provides enough additional features that (I hope) you can be persuaded to use it rather that your "pet" alternative. This section describes some common analysis systems, and shows how you might implement their structure using NOVA.

NOVA is (I think) fairly successful in this regard, but by providing such flexibility, it does allow you to build a "hybrid" system that is probably the *worst* of all worlds. The final section discusses a few of the more common pitfalls and how to avoid them.

1.Pure FORTRAN (or whatever language)

The USRn functions of NOVA allow you to do whatever you want with a minimum of extra effort. In particular, USRn functions have access to histogramming facilities and to all of the constants / variables defined by the user (a brief description of the FORTRAN interfaces provided is given at the end of this section). In this scenario, the *definition* part of NOVA consists of defining a single variable:

```
X = USR0 (X)
```

and the *OPSEQ* is the single line

```
EVAL X
```

This is equivalent to the one line FORTRAN program

CALL USR0 (X)

The obvious advantage of this approach is speed. NOVA is extremely flexible (allowing you, for example, to change variable definitions etc. "on the fly") but is certainly not blindingly fast! Typically (depending on complexity - there is more overhead for short expressions than for long ones), a *calculation* in NOVA might be 15 - 20 times slower than the equivalent FORTRAN program.

What you lose by going the "pure FORTRAN" route, of course, is the flexibility of being able to change things on the fly and the convenience of having NOVA tell you the "state of the system". If speed is more important than flexibility, this is the obvious route to take.

2. DACS

In DACS (and also HONDA for those of you who are *really old*), the logic of the analysis is contained entirely in the IF statements of the OPSEQ. NOVA allows this alternative, although some of the "Logic" may also be contained in Conditions placed on spectra. A second major difference with NOVA is that the definitions of variables need not be done first (and, of course, everything is done in "real time" - none of this waiting 45 minutes to compile an EPROC only to find out that you have made a silly typing mistake.

A simple DACS EPROC (containing only event types 1 (scalers) and 2) might be developed as follows.

```
1       (Insert things into OPSEQ)
        IF (~ EVTYPE1) GOTO EV2
        EVAL SCALERS
        GOTO $END_OPSEQ
EV2:    IF (~ EVTYPE2) GOTO $END_OPSEQ
        INCR SPID
        IF (PROTON) THEN
                INCR SX0
                INCR SY0
                INCR SX0Y0
                INCR SXF
        ENDIF
EOF     (Finish entering OPSEQ stuff)
```

Then you can go about defining things required to do these calculations.

```
CONDITION EVTYPE1, EVTYPE2
EVTYPE1 = $EVENTTYPE .EQ. 1
EVTYPE2 = $EVENTTYPE .EQ. 2

! Event Type 1 definitions
```

```
NUMSC=100
DELTIME=5.0
REAL*4 RATES(100), TOTS(100), OLDS(100)
PARAMETER TOTS,OLDS
RATES = USR0 (RATES,TOTS,OLDS,NUMSC,$RAW,DELTIME)

! Event Type 2 Definitions

DEF2D /XSIZE=100/YSIZE=100 SPID
DEF2D /XDATA=ESUM /YDATA=TTB SPID
DEF2D /XLOW=0/XHIGH=2047/YLOW=0/YHIGH=2047 SPID
ESUM = $RAW (10)
TTB = $RAW (15)
REAL*4 DRIFT_VECTOR (50)
DRIFT_VECTOR = USR1 (DRIFT_VECTOR, $RAW(DRIFT_OFFSET))
DRIFT_OFFSET = 39
DEF2D /XSIZE=50 /YSIZE=50 /XDATA=X0 /YDATA=Y0 SX0Y0
X0 = DRIFT_VECTOR (2)
Y0 = DRIFT_VECTOR (3)
```

etc.

Eventually you will get everything defined (you can ask NOVA at any time what is still undefined - LIST/UNDEFINED - and it will tell you).

3. LISA / PERSEUS / STAR

The architecture of many popular analysis systems is that you do all of you *calculations* in a user-written FORTRAN program, but the system increments spectra for you "automatically" based on conditions / tests applied to each spectrum. In NOVA, the OPSEQ corresponding to this could be:

```
1
        EVAL USER_FORTRAN
        EVAL $ALLCONDS
        INCR $ALLSPECTRA
        EOF
```

USER_FORTRAN calls the user FORTRAN routine to do all of the calculations, perhaps (not necessarily) placing all calculated variables in the vector USER_FORTRAN.

```
        USER_FORTRAN = USR0 (USER_FORTRAN)
```

Spectra are then defined with a *condition list* and are incremented by the statement INCR $ALLSPECTRA only if the conditions in the condition list for a spectrum are all TRUE.

```
        DEF1D /XSIZ=200 /XLOW=-45 /XHIGH=45 /XDATA=SCAT_ANGLE SANGLE
        DEF1D /CONDITION=(CHAMBER1_OK, CHAMBER2_OK) SANGLE
```

```
CHAMBER1_OK = INSIDE (XMIN, X1POS, XMAX)
CHAMBER2_OK = INSIDE (XMIN, X2POS, XMAX)
X1POS = USER_DATA (0)
X2POS = USER_DATA (1)
SCAT_ANGLE = USER_DATA (2)
```

Many of these systems include the concept of *FATAL* conditions (if *any* of a set of conditions is TRUE, processing of the event is immediately terminated). The OPSEQ which mimics this is:

```
1
        EVAL USER_DATA
        IF (FATAL) GOTO $END_OPSEQ
        EVAL $ALLCONDS
        INCR $ALLSPECTRA
```

FATAL is then defined to be a *condition group* containing whatever conditions you want to be fatal (note - there is no limit on the number of conditions in a group - the "maximum 128" conditions applies only to conditions in the spectrum *condition mask).*

```
ADDGRP /TYPE=CONDITION FATAL CHAMBER1_OK CHAMBER2_OK ...
```

A condition becomes a Fatal condition by just adding (ADDGROUP) it to the group FATAL, and is changed back to non-Fatal simply by removing it (SUBGROUP) from this group.

One of the more serious drawbacks to these systems (at least the one which people told me they would most like changed) is that there is only one "level" of FATAL condition - either something is declared FATAL or it isn't. NOVA easily allows any number of such levels, including the possibility of incrementing some spectra at each level.

```
1
        EVAL USER_DATA
        IF (FATAL_1) GOTO $END_OPSEQ
        INCR SPECGROUP_1
        IF (FATAL_2) GOTO $END_OPSEQ
        INCR SPECGROUP_2
```

As the names suggest, SPECGROUP_n are groups of spectra all of which will be incremented if the appropriate FATAL_n condition groups are FALSE. They would be defined as:

```
DEF1D /XSIZ=100 /XDATA=SCAT_ANGLE SANGLE
DEF1D /XSIZ=100 /XDATA=XPOS SXPOS

ADDGRP /TYPE=SPECTRUM SPECGROUP_1 SANGLE SXPOS
```

4. XSYS

The XSYS system is based on blocks of conditions / spectra each of which is governed by a single test - if it is FALSE the entire block is skipped. Again, all calculations are done in a user-written FORTRAN program. In NOVA:

```
1
        EVAL USER_FORTRAN
        IF (CONDITION_1) THEN
                EVAL CONDGROUP_1
                INCR SPECGROUP_1
        ENDIF
        IF ( CONDITION_2) THEN
                EVAL CONDGROUP_2
                INCR SPECGROUP_2
        ENDIF
```

5. "Pure" NOVA

The intent in writing NOVA is that the "User FORTRAN" part should be minimal (ideally zero, but there are some things that just aren't worth it). Especially when setting up an experiment, the convenience of being able to change any variable in the system at any time is immense (at least I think it is), and you lose this flexibility when lots of your code is buried in a user FORTRAN routine. For example, suppose you are "tuning" the resolution on the MRS. You start with a "simple" definition of focal plane position:

```
        XF = (VDCSEP*X1-F*DX12)/(VDCSEP-DX12*TAN_DELTA)
        TAN_DELTA=-.04
        VDCSEP = 5440
        DX12 = X1 - X2
        X1C = X1 + X1C0
        X1C0 = 7420

        XF_CORR = XF
```

You then look at correlations between XF_CORR (=XF so far) and other MRS variables, and decide that the focal plane position should be corrected for THETA. Simply re-define *on the fly:*

```
        THETA = 0.5*((VDCDIST/DX12) - 1)
        XF_CORR = XF + A*THETA**2
```

and your XF_CORR spectrum now contains the additional term. You can then adjust the coefficient A to give you the "best" resolution

```
        A = 100
(have a look at the spectrum)
        A = 120
etc.
```

and then carry on to look at other aberrations.

FORTRAN Interface

NOVA provides several functions which allow user-written FORTRAN routines (i.e. USRn functions) access to the variables / parameters / spectra defined in the NOVA tables.

First of all, the user routine must have access to the tables themselves. This is accomplished by passing to the USRn function the argument $TABLES. For the remainder of this section, let's assumed that you have defined a USRn function with the statement

X = USR0 (X, $TABLES)

The FORTRAN subroutine (remember that it should be written as a *subroutine*, not a Function) begins with the statements:

```
SUBROUTINE USR0 (X, NOVATABLE)
INTEGER*4 NOVATABLE (0:1)
```

(note that the index of the table starts at 0, not 1). Let's also assume (for simplicity) that this routine just stores things directly in the tables - the "return value" X is not used.

1. Accessing parameters

One thing you probably want to do is to access parameters stored in the tables (e.g. constants like CHAMB1_DIST = 32). Assuming that this thing really is a (pre-calculated) parameter (more about variables in a minute), all you need is the place in NOVATABLE where its *value* is stored. You get this from the function

ICHPTR = N_VALPTR (NOVATABLE, 'CHAMB1_DIST')

This sets the variable ICHPTR to the index in NOVATABLE where the value of CHAMB1_DIST is stored. You can then get at the value with

IDIST = NOVATABLE (ICHPTR)

Alternatively, suppose that CHAMB1_DIST is something that is calculated by your USR0 routine, and you want to store the value there for someone else to use. Then you just say

NOVATABLE (ICHPTR) = IDIST

(A technical note - when CHAMB1_DIST is defined to NOVA, it should be declared a PARAMETER, since NOVA doesn't have any formula for calculating its value. It is then the user's responsibility to ensure that his USR0 routine is called and the value set before anyone else in NOVA tries to use it).

If CHAMB1_DIST is a Floating Point rather than an Integer value, things are a little more complicated (it is assumed throughout that you know what kind of values you are dealing with - INTEGER*4 or REAL*4). REAL*4 values are stored "as is" in NOVATABLE, so you have to trick the FORTRAN compiler into giving you the value stored in (the Integer) NOVATABLE (ICHPTR) as a Floating Point value. You do this using the EQUIVALENCE statement

```
EQUIVALENCE (DIST, IDIST)
        .
        .
IDIST = NOVATABLE (ICHPTR)
```

The Floating Point variable DIST now contains the value that you want. Note that you MUST NOT say:

```
DIST = NOVATABLE (ICHPTR)
```

since this will take the (assumed) INTEGER*4 value and convert it to the same value in Floating Point - not at all what you want (if this isn't clear, don't worry about it - just do it this way and take my word for it that the other way won't work).

2. Accessing Variables

Variables in NOVA are things that depend on other things (either $RAW data or other variables) and that NOVA *knows how to calculate* (i.e. it has been given a "formula" for evaluating the variable). If you want to use the value of one of these, you have to ask NOVA to do the calculation for you (if you are *ABSOLUTELY SURE* that NOVA has calculated the variable already earlier in the OPSEQ, you can use the same procedure as above for PARAMETERS, but this is not recommended - you might change the OPSEQ so that this assumption is no longer valid).

For variables, NOVATABLE contains an internal representation of the formula (a so-called "evaluation list") which tells it how to calculate the variable. The function

```
LISTPTR = N_EVPTR (NOVATABLE, 'variable_name')
```

sets LISTPTR to the address of this evaluation list in NOVATABLE. To "execute" this list and calculate the value of the variable, use the functions:

```
IVALUE = N_EVAL (NOVATABLE, LISTPTR)
XVALUE = FP_EVAL (NOVATABLE, LISTPTR)
```

Returning to the example above, if CHAMB1_DIST were a (Floating Point) variable instead of a PARAMETER, you could get at its value by saying

```
LISTPTR = N_EVPTR (NOVATABLE, 'CHAMB1_DIST')
DIST = FP_EVAL (NOVATABLE, LISTPTR)
```

Again, it is assumed that you know what kind of variables you are dealing with - INTEGER or REAL. Don't use N_EVAL for a Floating Point variable - the answer will be wrong.

If the variable is actually a Condition, N_EVAL (or FP_EVAL) will update the software scalers automatically for you. Also, these routines are clever enough to know whether or not they have already calculated the variable for this event - they will only do it once. Thus, if you were to write:

```
LISTPTR = N_EVPTR (NOVATABLE, 'CHAMB1_DIST')
DIST1 = FP_EVAL (NOVATABLE, LISTPTR)
DIST2 = FP_EVAL (NOVATABLE, LISTPTR)
DIST3 = FP_EVAL (NOVATABLE, LISTPTR)
```

the "evaluation list" stored at LISTPTR would only be executed once (the first time) (and for a condition, the software scalers would only be incremented once). The value returned would be correct in all instances, however.

3. Spectrum Incrementing

The function

```
ISPECPTR = N_SPECPTR (NOVATABLE, 'spectrum_name')
```

returns the index in NOVATABLE of a "spectrum descriptor block" for the spectrum_name'. This index is then passed to the subroutine SPINCR to increment the spectrum.

```
ISX1PTR = N_SPECPTR (NOVATABLE, 'SX1')
CALL SPINCR (NOVATABLE, ISX1PTR, WEIGHT)
```

where WEIGHT is the (Floating Point) value to be added to the appropriate channel of SX1 (typically, WEIGHT = 1.0).

4. USRSETUP

As you might expect, routines like N_VALPTR are not blindingly fast - they have to search through all the named variables defined in NOVA to find the position of the one that you want. Not the sort of thing that you want to do on every event! (the whole point of this was to make it *fast*, after all).

Since the position of things in NOVATABLE doesn't change while you are analyzing (the system automatically PAUSES analysis when the tables are changed and things might move around), you should be able to determine the positions of such named variables "once and for all". This is where the routine USRSETUP comes into play.

This routine is called by NOVA every time you start analysis (the ANALYSE or EA command), so you can put your calls to N_VALPTR, N_SPECPTR etc. in here, and pass the results (via a COMMON area) to your USRn routines which are executed for every event. If you change the tables (e.g. define something new), these pointers might change, but NOVA will PAUSE analysis for you, and when you start it up again with the ANALYSE command, USRSETUP will get called again and you will get the new positions of these variables.

So USRSETUP might look something like:

```
      SUBROUTINE USRSETUP (NOVATABLE, *)
      INTEGER*4 NOVATABLE (0:1)
C
      INTEGER*4 ICHPTR, LISTPTR, ISPECPTR
      COMMON /USER_PTRS/ ICHPTR, LISTPTR, ISPECPTR
C
      ICHPTR = N_VARPTR (NOVATABLE, 'CHAMB1_DIST')
      LISTPTR = N_EVPTR (NOVATABLE, 'CHAMB1__DIST')
      ISPECPTR = N_SPECPTR (NOVATABLE, 'SX1')
C
C     USRSETUP takes the alternate RETURN if something goes wrong
C     Assume the LOGICAL variable OK is set .FALSE. if problems occur
C
      IF (.NOT. OK) THEN
            RETURN 1
      ELSE
            RETURN
      ENDIF
      END
```

All of these functions (N_VALPTR, N_EVPTR ...) return a *negative value* if the specified variable/ spectrum doesn't exist. USRSETUP could check all of these and take the alternate return (RETURN 1) if your NOVA tables are incomplete. The system will issue the error message 'Analysis aborted by User set up routine' - if more detailed information is desired, USRSETUP should write a message to the terminal (TYPE or WRITE (6, ...)). Alternatively, your USRn functions could simply check that the pointers it has been given are reasonable before it tries to use them.

```
      SUBROUTINE USR0 (X, NOVATABLE)
      INTEGER*4 NOVATABLE (0:1)
C
C     These pointers are calculated in USRSETUP whenever you enter
C     the ANALYSE or EA command
C
      INTEGER*4 ICHPTR, LISTPTR, ISPECPTR
      COMMON /USER_PTRS/ ICHPTR, LISTPTR, ISPECPTR
C
C     Event Analysis
C
      IF (ICHPTR .GE. 0) IDIST = NOVATABLE (ICHPTR)

      etc.
```

5. Other Functions

This is all that has been implemented as far as 'user-callable' NOVA functions is concerned. I could provide things that let you define new histograms in your USRSETUP routine for example (bypassing the DEF1D command), although the error checking /

recovery would then have to be provided by the user, so I am a little reluctant to do this. If there is anything else that you would like to see made available to the user, let me know and I will consider it.

SCALERS Processing

Just because it is common to virtually all experiments, NOVA provides a default USR0 routine to do Scalers processing.

First of all, note that this routine assumes:

1. Your data was written with VDACS.
2. All scalers are 24 bits long.
3. All scalers were read using TWOTRAN "block read" functions - CFQIGNORE etc. (which include a block header word indicating how long each block is).
4. There is only one scaler event (the event type can be whatever you like) and this event contains *only scalers* (i.e. the first word in the event is one of the "block headers").

If you are in the process of designing your TWOTRAN program, you might want to keep these points in mind. If your scaler data is not of this format, you are stuck with writing your own routine to process them.

Note that these routine assume that *they will get all scaler events to process.* If they don't, the overflow calculation will be wrong, and the RATES calculation (which assumes that successive scaler reads are always separated by the clock interval) will also be wrong. You should ensure this in your TWOTRAN program by specifying that scaler events are MUSTPROCESS events. (A word of warning. There are some indications that this MUSTPROCESS statement in TWOTRAN may not work reliably. If you suspect that this is the case, get in touch with someone from the VDACS support group and show them the problem).

The USRn function provided with NOVA to process scalers is USR0. If you have your own USR0 routine which does something else, you can include it in your NOVAUSR.OLB and it will over-ride the default.

You must define the following things for NOVA.

```
!       Time interval between scaler reads (this is used to calculate RATES in
!               count/second rather then counts / N seconds.
REAL*4 DELTIME
DELTIME=5.0
!       Number of scalers ins the scalers event
NUMSCL = 100
REAL*4 RATES(100), TOTS(100), OLDS(100)
PARAMETER TOTS, OLDS
RATES = USR0 (RATES,TOTS,OLDS,NUMSCL,$RAW,DELTIME)
```

and in the OPSEQ (assuming scalers are event type 1, although this is not required)

```
IF ($EVENTTYPE .EQ. 1) THEN
        EVAL RATES
ENDIF
```

The routine puts the accumulated total scalers in the vector TOTS and the rates (the difference between the last two scaler events divided by DELTIME) in the vector RATES. You can then (from within NOVA)

```
LIST/FULL RATES
LIST/FULL TOTS
```

to see the current values.

I have also provided two programs - NRATES and NSCALERS - that allow you to look at the scalers (nicely formatted) from VAX DCL. Note that if you want to use these programs, you *must* use the names given above (at least you must use the names TOTS and RATES - the others are arbitrary). Use of these programs is exactly the same as those currently in use on the MRS data acquisition system (they are described in a separate manual) - they are "special" only in the sense that they take their input from a NOVA subprocess rather than reading the input stream (GRA1:) directly.

PITFALLS

1. Double Logic

The most obvious pitfall in NOVA (at least to me) is that there is a "parallel logic" built into it. That is, whether or not a spectrum is incremented depends on two (sort of) unrelated things:

i) The *CONDITION LIST* associated with the spectrum.
ii) The position of the *INCR* statement in the OPSEQ.

As a simple example, suppose you define:

```
DEF1D/COND=C1/... SX

1
IF (C2) THEN
        INCR SX
ENDIF
^Z
```

You then look at SX and see that it isn't being incremented very often, yet the condition C2 is almost always TRUE. Why?

The problem, of course, is that the condition C1 (in the Condition List for SX) *also* must be TRUE for SX to be incremented, and you (the user) have to be clever enough (or rather have a good enough memory) to recall that C1 is in the condition list for SX. There are *TWO* conditions on the spectrum SX, but they appear on different places (one in the OPSEQ and one in the condition list), and it is all too easy to forget one of them when you are busy looking at the other one (not too tough in this example, but think about an experiment with several hundred spectra and a complex OPSEQ which was created 6 months ago when you last ran this experiment).

The "solution" to this problem is, obviously, *don't do it.* Both the Logic Tree of the OPSEQ and condition on spectra have their place, but their place is probably not together! Pick one method of placing condition on spectra, and stick to it.

NOVA helps you out a little bit in this case. While there are two conditions on this spectrum, there are software scalers associated with each one that help you sort out what is going on.

In the above example, the software scaler on the spectrum SX would tell you that NOVA "tried" to increment the spectrum exactly the same number of times as the condition C2 was TRUE, but was "successful" a smaller number of times. This should be a clue that it is the condition list on SX which is causing the problem. In the other scenario (where you are looking at C1 and wondering why the number of counts is so small), the clue is that the number of time NOVA "tried" to increment the histogram is too small, and this should point you to the fact that it never got to the INCR statement in the OPSEQ often enough.

Nonetheless, I admit that the potential for confusion exists, and it requires a certain amount of "cleverness" on the part of the user to sort it out. It is probably best to avoid such "parallel logic" schemes, and stick to one method or the other.