TRIUMF	UNIVERSITY OF ALBERTA EDMONTON ALBERTA	
	Date 2-March-1991	*File No* TRI-DNA-91-1
Author P. W. Green		*Page* 1
Subject Introduction to the NOVA Data Analysis System		

NOVA (Version 1.300)

This note contains a very general introduction to the NOVA data analysis system. If you are new to NOVA, this should be the first document that you read. Companion documents describe the Commands and the Operations Sequence (OPSEQ) in greater detail. I will attempt (probably unsuccessfully) to keep up-to-date copies of this documentation on the TRIUMF cluster in NOVA$DIR.

The NOVA system is under development, and what is described here is either (i) what the system now does, or (ii) what I hope it will do when it is complete (be forewarned - it is not always clear into which category things fall). As far as I know, everything that is planned for NOVA is possible within the current implementation, but there is always the possibility that something will turn out to be either not possible or extremely difficult. Therefore, there are no guarantees about anything that you see here, other than the fact that I will do my best.

As with any new system, there will undoubtedly be bugs / suggestions for improvement / whatever. Please address all complaints and /or suggestions to Peter Green (PEWG@TRIUMFCL of SARCEE::PEWG). If the problem is drastic (i.e. you are losing beam time because of it), I can usually be reached at the University of Alberta (403-492-3637) if I am not at TRIUMF (local 478).

Overview

NOVA is a system designed to aid in the analysis (either on-line or off-line) of experimental nuclear physics data. It receives as input (either from a data file - magnetic tape or disk, for example - or from a data acquisition process such as VDACS) an "event" consisting of *raw* data variables. These are combined in user specified ways to produce *calculated* variables (presumably, something closer to the physics that you are trying to extract from the experiment). Finally, histograms of either raw or calculated variables are produced.

Specification of how the calculations are to be performed to produce these histograms can be divided into three sections.

1. **Declarations** The system must be told *what kind* of object each variable is (INTEGER or REAL for example). This is done with data declaration statements, analogous to type declaration statement in FORTRAN (INTEGER*4, REAL*4 etc.). The standard FORTRAN defaults apply to variables which are not explicitly declared (variables beginning with I, J, K, L, M, N are INTEGER*4, all others are REAL*4).
2. **Calculations** For every variable, the system must be told *how* to calculate its value. The expressions defining how a variable is to be calculated are entered using standard FORTRAN syntax.

3. **OPSEQ - OPeration SEQuence or Logic Tree** This is the *program* which is executed for each event given to the system, and deals (mainly) with what histograms are to be incremented (and under what conditions).

Each of these sections is discussed in greater detail below.

The aim in designing NOVA was to provide a system in which the user could specify his analysis in a "top-down" fashion. Most people (I think so, anyway) find it easier to begin with the higher levels of the program and work down to the details. In the case of NOVA, the user can begin by thinking about histograms and the order in which they should be incremented. After this is done, he can proceed to thinking more carefully about what variables are required for these histograms, and how they are ultimately related to the raw input data from the event.

This order is not enforced, however. If you are more comfortable starting with raw data variables and working "up" to more complicated calculated variables and (eventually) to histograms, that is perfectly acceptable.

Another important goal of the system was flexibility. The user should be able to change *anything* in the system "on the fly". Especially during the setting up and debugging phase of an experiment, it should be possible to redefine either the definition of a variable (either parameters or which variables are variables included in the definition), histograms and conditions (Logic Tree) and see the result of the changes immediately ("immediately" being somewhat arbitrarily defined as less that 15 seconds - preferably *much less*.

I think that both of these goals have been adequately met in this initial version of the program. Statements can be inserted into or deleted from the Operation Sequence at will. Variables can be re-defined at any time. Individual spectra can be deleted at any time and any parameter in a spectrum definition (limits, which variable is being histogrammed, etc.) *except the spectrum size* (I'm working on that one) can be altered at any time.

A word of warning about the above statement is in order. It turned out to be very tricky to re-define the *type* of a variable (that is, once a variable got to be REAL*4, either explicitly or implicitly, how do you change it to be INTEGER*4). I am pretty sure that I have solved this problem, but if you start getting funny results, try deleting the variable, explicitly declaring its type first and then re-entering its definition. If this cures the problem, please write down the details (and save ALL appropriate data files - see the section on Crash Recovery below) and give them to me, so I can fix it.

Using the Program

NOVA consists of two main parts. First, there is a sub-process which does the actual data analysis - reads events from the input stream, does the arithmetic manipulations to produce calculated variables and increments the histograms. In normal operation, this sub-process runs "in the background" all the time, with very little user intervention.

Second, there is a user process which interacts with this sub-process whenever the user wishes to change things (e.g. start or stop a run, display spectra, change definitions, etc.). This mode of operations should be familiar to most LISA users.

A word of warning. The phrase *NOVA* in this document is used to refer to the program that you most often use to send commands to the sub-process (usually), the sub-process itself (occasionally) and the complete system (rarely). I hope that the context makes it clear which aspect of *NOVA* is being referred to in each case.

In addition to these two most often used programs, there are a few auxiliary programs which can be run to perform specific functions. These are described in this section. *Note that all commands in this section are entered as VMS DCL commands (i.e. you must be talking to VMS when you enter these commands).* In this document, the program name is prefixed with the usual $ prompt that you get from VMS DCL to remind you that this is a VMS command word. Most other commands that you use to interact with the data analysis sub-process are entered by first running *NOVA* - you will then be talking to the NOVA command line interpreter.

$ **NOVASTART** **[name]**

This command is used to start up a sub-process. The *name* parameter is optional - if it is not given, the system will substitute the process ID of the created sub-process for *name.* The actual sub-process name will be *NOVA_name.* Since VMS process names are limited to 15 characters, the *name* argument (if entered) must be no more than 10 characters long.

It is possible to have several NOVA sub-processes active at the same time. Only one of them is "current" at any one time, however (all of them can be busy analyzing data concurrently, but you can only talk to one of them at a time - see the *MAPTO* command below). If your process already owns (i.e. has started) another NOVA sub-process, this command will ask if you wish to start another one. The number of such sub-processes is limited only by the restrictions imposed by the VMS operating system.

$ **NOVAKILL** **[name]**

This command is used to terminate a NOVA sub-process. If *name* is not given, the system will list the NOVA sub-processes currently owned by you (unless there is only one) and then request verification that you really want to delete it. Note that you must have write access to a particular NOVA sub-process in order to delete it (see the LOCK / UNLOCK commands). Also note that all of your definitions / spectra will be lost unless you have DUMPed them to disk.

$ **NOVAWAIT**

This command is useful mainly in batch jobs or command files, and simply waits until an end-of-run condition is detected for the current NOVA sub-process (before dumping the spectra to a disk file, for example).

$ **MAPTO** **[name]**

This command selects which NOVA sub-process you want to talk to when you run the NOVA program (e.g. to display histograms). You are currently allowed to *MAPTO* any NOVA sub-process on the system (whether it belongs to you or not!). This allows, for example, someone to sign on at any terminal (even remotely via modem) and have a look at the progress of his experiment.

If you are talking to the NOVA command line interpreter (i.e. running the NOVA program), the STAT command will tell you which NOVA sub-process you are currently mapped to. If you are talking to VMS DCL, $SHOW LOGICAL NOVA$PID will give you the process ID of the current NOVA sub-process.

In principle, you can *MAPTO* any NOVA sub-process on the system (whether it belongs to you or not, although standard VMS UIC-based protections still apply). I have implemented a software check within NOVA which allows only one process (the "owner") to *change* a particular sub-process (see the LOCK / UNLOCK commands), so I <u>believe</u> that it is not possible for you to alter someone else's NOVA (define new variables / histograms, for example). This protection scheme has not been thoroughly tested, however, so **<u>please</u>** be careful when you are fiddling with someone else's experiment.

$ **<u>NOVA</u>**

This command invokes the NOVA program (which has its own command line interpreter) and allows you to interact with the current NOVA sub-process (set with the **<u>MAPTO</u>** command if there is more than one alternative) to examine / modify the parameters of the analysis. It is described in more detail later (and also in a separate document - NOVA commands).

<u>CRASH RECOVERY</u>

Neither NOVA nor users is bulletproof. Either through a careless typing error or through a bug in the program (I'm not bulletproof either) you can end up with a set of definitions that are either gone or hopelessly scrambled (any you were just about to do your backup too). What can you do?

NOVA protects you (at least a little bit) and if you are careful, you can probably recover (if you are careful *after* you make the error - we already know you weren't careful beforehand).

NOVA maintains two disk files. The first (NOVA_BACKUP.TBL) contains a copy of your tables at the beginning of this session (i.e. just before you execute a command which will change them - that's why the first command in a NOVA session sometimes takes a couple of seconds to execute - it is writing to the backup disk file). The second file (NOVAINPUT.LOG) contains every command that you entered during the session (@-files are expanded - that is the file doesn't contain the string *@filename* but it does contain all the commands that were executed when the @-file was executed).

So (in principle at least) you can get back to exactly the same situation that you are now in as follows:

1. CTRL-Y out of NOVA *(don't use the EXIT command - this deletes the backup file).* You could use EXIT/SAVE if you really want to, but CTRL-Y is probably easier to remember.

2. RENAME NOVAINPUT.LOG someotherfilename

(3. See below)

4. NOVA/RECOVER
(or just NOVA - it will recognize that the backup file is present and ask you if you want to recover - say YES).

5. @someotherfilename (the one you generated in step 2).

This gets you back to exactly where you were before (including the stupid typing mistake you made), which is probably <u>not</u> exactly what you want. Therefore, we should include step #3 in the above:

3. Edit someotherfilename and remove the offending statements (e.g. the DELETE * that you didn't really mean to type). This allows you to save what you want from the previous session but get rid of the silly mistake.

So to summarize, when you realize that you have just done something silly in NOVA, the following sequence of commands should be executed <u>immediately</u>.

^Y
$ RENAME NOVAINPUT.LOG savefile.dat
$ EDIT savefile.dat
.
. editor commands to remove what you don't want
.
$ NOVA/RECOVER
Nova> **@savefile.dat**

<u>Please Help Me</u>

There is an ulterior motive in all of this. In the (extremely unlikely, I admit) case where the crash is *my fault* (perish the thought), it would be invaluable for me to be able to re-create the problem. Therefore, if you suspect that I am the culprit, ***<u>please!:</u>***

1a). COPY NOVA_BACKUP.TBL 'yetanotherfilename'

6. Save both 'someotherfilename' and 'yetanotherfilename', so that I can re-create (and therefore fix) the problem. If you LOADed a file (i.e. if 'someotherfilename' contains any LOAD commands), save them too.

<u>Personalized NOVA</u>

If you don't need anything out of the ordinary, then the standard system-supplied NOVA will probably do everything that you need and you can ignore this section.

If, however, you have (for example) your own specific USRn user functions, then you will need to link a specialized version of the NOVA subprocess containing them. This section describes how you go about this.

First (in whatever default directory you plan to use to build your specialized version of NOVA), you must

$ LIB/CREATE yourlibrary.OLB

Second, you must tell the LINKNOVA command procedure that you have a special library that contains your programs / subroutines.

$ DEFINE NOVAUSR DISK:[yourdirectory]yourlibrary.OLB

Yourlibrary.OLB is a VMS object library, which will contain all of your user-specific code (typically, all or your USRn routines and perhaps a USRSETUP, along with any subroutines which they require). When you compile your subroutine, you must insert the object module into this library

$ FORTRAN yourfile
$ LIB/REPLACE NOVAUSR yourfile

Then, to build your own customized version of NOVA.

$ LINKNOVA

To run the new version of the program after you have built it.

$ NOVAKILL (the old one, if it is still present)
$ NOVASTART (to start up the new sub-process)
$ NOVA (to load the NOVA command line interpreter and start defining variables / histograms etc.)

Names

Everything in NOVA (variables, conditions, histograms, etc.) is referred to by *name.* (many popular analysis systems refer to things - especially spectra - only by number. I find this very confusing, so I decided that things should always be referred to by a name which is long enough that you have a reasonable chance of being able to remember what it is from the name only. If you *really* like the number idea, you can use descriptive names like S001, S002 etc. for your spectra.)

A name is a sequence of from one to twelve characters consisting of digits (0-9), letters (A-Z), dollar sign ($) and underscore(_). *Upper and lower case letters are equivalent - all input to the NOVA command line interpreter is effectively converted to Upper Case.* The first character of a name must be either a letter or a $. Note that *all names in the system must be unique.* (many systems allow duplicate names - for example a variable and a spectrum of that variable. Not in NOVA - the names must be different).

Variables

The most common use of a name in NOVA is to define a variable. As in FORTRAN, every variable has a *type* associated with it - the recognized variable types are INTEGER*4, INTEGER*2, REAL*4 and CONDITION. (A *condition* is somewhat analogous to a logical variable in FORTRAN - conditions are described in more detail later). In NOVA, a condition is primarily intended to be used as a test (e.g. should a histogram be incremented, should I execute this portion of the OPSEQ) and always has associated with it a *software scaler* which keeps track of how many times the variable was calculated and how often it was found to be "true". The value associated with a condition is always of type INTEGER*4.

The *type* of a variable is established either explicitly through a declaration statement, or implicitly whenever the variable is first used in a definition. Implicit variable typing follows the usual (slightly modified for the $ character) FORTRAN rules:

1. If the first character of the name is **I**, **J**, **K**, **L**, **M**, **N** or **$**, the variable is assumed to be INTEGER*4.

2. Otherwise, the variable is assumed to be REAL*4.

Variables may be re-defined as conditions (and vice-versa) at (almost) any time.

Variables (but not conditions) may also be dimensioned (single subscript only). Subscripts *always start at zero.* This is not (yet) handled very nicely in the "default" (i.e. non-explicit declaration) case (a variable is "implicitly" dimensioned if the first reference to it is an indexed reference - the actual size must be established in a subsequent declaration statement). It is safest (for now) to explicitly dimension any variables you want <u>before</u> you use them.

Certain names are pre-defined in NOVA. These include:

$ALLCONDS A condition group (groups are explained below) containing all conditions.

$ALLSPECTRA A spectrum group containing all spectra.

$CONDMASK The spectrum condition mask. This is described in more detail below.

$EVENTLENGTH This is the event length in either words or longwords (depending on the type of the variable **$RAW.**

$EVENTTYPE The event type.

$FALSEMASK Another part of the condition mask. Also described below.

$RAW (N) This is the raw event data to be analysed. By default it is an **INTEGER*2** array of length 512 words - the type and /or maximum length can be changed explicitly if desired. The first element (**$RAW(0)**) is the first *data* word in the event. Event headers (event sequence number, event type, etc.), if they are present, are stripped off and stored elsewhere. The actual length of this vector is specified by the variable **$EVENTLENGTH**). If the length of the input event exceeds the maximum declared for **$RAW**, the excess data is simply lost (i.e. is not available to the analysis process).

$RUNNUMBER The run number.

$TABLES This is a variable which contains the entire table structure of NOVA (all variables, spectrum definitions, etc.). It is used primarily as an input argument to *user functions* (described later). Routines are available to access any element in **$TABLES** from within such user functions.

Of these, user programs will most often make use of the event type (to skip over various parts of the operation sequence for different event types) and the raw data vector (all calculated variables are ultimately defined in terms of the raw data vector). In some instances, the two groups **$ALLSPECTRA** and **$ALLCONDS** might also be useful (see the examples following the description of the OPSEQ for ways in which these might be used).

Declarations

Variables are *declared* much as in FORTRAN. Assuming that you are talking to NOVA (i.e. the NOVA CLI, not VMS), just enter a type declaration keyword (**INTEGER*4**, for example, or any unique abbreviation such as **INT*4, I*4**) followed by a list of the variables which are to be that type. Variable names in the list can be separated by commas or (one or more) blanks or tabs. Examples of valid declaration statements are:

```
INTEGER*4 X, Y,    Z
REAL*4 A   B , C (20)
INT*2 $INT2_VAR
CONDITION X_GT_Y
DIMENSION IV(10)
```

Remember that, unlike FORTRAN, you don't have to enter all of these at the beginning of your session. You can enter declaration statements at any time.

Definitions

A variable is *defined* by telling the system how to calculate its value. Such definitions are entered using ordinary FORTRAN syntax (at least as close to "Standard" FORTRAN as I could get without writing a full blown compiler. There may be some differences - i.e. some perfectly legal FORTRAN expressions which are not accepted by NOVA - but I don't think you are likely to run into them. If you do, give me the details and I will try to include it in the next release of the system).

For example, a variable **X** might be *defined* by entering the command:

X = 14*(Y1-Y2)/(Y1+Y2+137.)

Mixed mode expressions (i.e. combinations of REAL and INTEGER values) are allowed (with a few exceptions as noted below). Expressions may also contain LOGICAL operators, as in

LOGICALVAR = (X .GT. Y) .AND. (Y .LE. 1)

Logical values (i.e. the results of Logical expressions, such as the variable **LOGICALVAR** above) receive the (INTEGER*4) value 0 if the expression is FALSE and -1 (all bits set) if the expression is TRUE.

NOVA allows you to enter INTEGER constants in Decimal (the default), Octal ('nnnn'O) or Hexadecimal ('nnnn'X).

Variables can be defined in any order, and may be re-defined at will. For example, when entering the expression for X above, the variables Y1 and Y2 need not yet be defined. Their definitions may be entered at a later time (although all variables must eventually be defined in terms of something - ultimately in terms of either the input event **$RAW** or parameters). Undefined variables do not (I think) cause things to crash - they just assume "garbage" values.

A *parameter* is defined simply by entering the expression as a single number

Y1 = 147.35

A *parameter* is simply a special kind of variable (one that doesn't have to be "calculated" - its value is already known). A variable may be re-defined as a constant (and vice-versa) at any time.

A vector of parameters may also be defined (it might be used, for example, to pass a whole table of parameters to a user function). In this case, the vector must be both dimensioned and explicitly declared as a parameter (using the PARAMETER command). Only then can individual elements be set. For example:

REAL*4 GAINS (20)
PARAMETER GAINS
GAINS (0) = 1.13
GAINS (1) = 1.27
etc.

While analyzing an event, NOVA will calculate variables *as it needs them* - the order in which the definitions were entered is immaterial. One potential problem with this, however, is that circular definitions are possible, and the system can't (yet) detect them. As a simple example, one could enter the two statements:

A = B
B = A

The method of evaluating variables guarantees that the program will not "hang" in this case - however, the values for both A and B will be garbage.

Standard FORTRAN precedence is used in evaluating expressions, and expressions can be of arbitrary complexity (not *really* arbitrary, but you are unlikely to hit the limit). When two operators are of equal precedence, they are evaluated left to right (e.g. A - B + C is evaluated as (A - B) + C)) exactly as in FORTRAN (except for exponentiation, which, also as in FORTRAN, is done right to left - A**B**C = A ** (B ** C)). Operators in order of precedence (highest to lowest) are:

9	-	Unary minus (as in -X + 2)
	+	Unary plus
8	**	Exponentiation
7	*	Multiplication
	/	Division
6	+	Addition
	-	Subtraction
5	.NOT.	
	~	Logical NOT (complement all bits)
4	.AND.	
	&	Logical AND
3	.OR.	
	|	Logical OR
2	.XOR.	
	%	Logical exclusive OR
1	.GT.	
	>	Greater than
	.GE.	
	>=	Greater or Equal
	.EQ.	
	==	Equal to
	.NE.	
	~=	Not Equal to
	.LE.	
	<=	Less than or Equal
	.LT.	
	<	Less than

Note that there are equivalent "standard FORTRAN forms" (.LE.) and more natural (I think) forms (<=) for all logical and relational operators. For you **C** programmers in the crowd, things like **&** for **.AND.** and **|** for **.OR.** should be familiar.

The logical operators (.AND., .OR., .XOR.) require some further explanation. These operate on INTEGER values only (attempts to use REAL variables cause a syntax error - one of the places where mixed mode is not allowed) and do a Bitwise AND (for example) on the two values. This might be used, for example, to MASK an INTEGER variable. However, in a logical TEST (e.g. an IF statement, or in a condition), *a value is TRUE if it is <u>any</u> non-zero value, and FALSE only if it is zero.* For example, if I=1 and J=2, both I and J would be TRUE if tested as a logical variable (since they are both non-zero), but the expression **(I&J)** would be FALSE since the bitwise AND of 1 and 2 is 0. A little confusing, certainly, since, as a Logical expression this says that TRUE (1) .AND. TRUE (2) = .FALSE.! This problem does not occur when the variables are really logical variables (e.g. the result of relational operations), since then TRUE is represented as -1 (for example, X>4 & Y<3 works as expected).

Variables can also be dimensioned (single index only), and the index must be of type INTEGER (no mixed mode allowed here, either). Indices always start at 0 (i.e. the first element in the vector **X** is **X(0)**, not **X(1)** as in FORTRAN. Individual elements in a vector cannot be set with an assignment statement unless the vector is first explicitly declared using the PARAMETER statement.

Finally, there are a limited number of *functions* available which can be used in expressions. They include (where **R** represents a REAL*4 value and **I** an INTEGER*2 or INTEGER*4 value):

SQRT (R)	
SQRT (I)	Square root
EXP(R)	
EXP(I)	Exponential
LOG (R)	
LOG (I)	Natural logarithm (base e)
COS (R)	
COS (I)	Cosine (angle in Radians, as in FORTRAN)
SIN (R)	
SIN (I)	Sine (angle in Radians)
INSIDE (ILOW, I, IHIGH)	
INSIDE (RLOW, R, RHIGH)	Logical function which is TRUE (-1) if RLOW <= R <= RHIGH (i.e. R is **inside** the window (RLOW, RHIGH). Note that *all 3 values* must be the same type (either INTEGER or REAL - no "mixed mode" allowed here either, so INSIDE (1, R, 2) would result in a syntax error - you have to use INSIDE (1., R, 2.)), and that the INSIDE function is **Inclusive** (TRUE if the variable is equal to either of the limits).
OUTSIDE (ILOW, I, IHIGH)	
OUTSIDE (RLOW, R, RHIGH)	Logical function which is TRUE (-1) if R < RLOW or R > RHIGH. Again all 3 values must be the same type and the OUTSIDE function is **Exclusive** (FALSE if the variable is equal to either of the limits).

All of these functions return a *REAL*4* result (regardless of the type of the argument) except for the last two (**INSIDE** and **OUTSIDE)** which return a Logical (i.e. INTEGER*4) value - 0 if the function is FALSE and -1 if it is TRUE.

As well, a small number of *User Functions* are recognized. The format is:

USRn (OUTPUT, INPUT_1, INPUT_2, ...)

where **n** is a single digit (0 - 9). User functions are provided to deal with those situations which require the full power of FORTRAN (or any other language). An obvious example is drift chamber decoding. There may be any number (up to 9) and any type of arguments passed to these functions. NOVA performs *no checking of these arguments* (either number of arguments or type), so it is the user's responsibility to ensure that the number and type of arguments agrees with the FORTRAN routine which will be called to perform the function. The routine need not be written in FORTRAN - any language will do provided that it obeys the FORTRAN convention for passing arguments. Arguments are passed by *address* (the term often used is by reference) not by value as, for example, in C.

The FORTRAN routine called to execute this User function should begin with a statement like:

SUBROUTINE USR0 (OUTPUT, INPUT_1, INPUT_2, ...)

(note that, although it looks like a *function* in NOVA, the routine should be coded as a subroutine in FORTRAN. If it is a FORTRAN FUNCTION, ***the return value will be ignored***!)

As the names suggest, the first argument to the function is assumed to be an *output* argument (i.e. something which is calculated *by* the user function and returned to NOVA) - the remaining arguments are assumed to be *input* arguments (things which NOVA will attempt to calculate from its own definitions before calling the User function). Of course, if any of these "Input arguments" is declared as a PARAMETER, then NOVA thinks that its value is already known and hence will not attempt to calculate its vale. Any of these arguments may be either a simple variable or a vector - it is again left up to the user to ensure that the assumptions of the usage within NOVA and the FORTRAN program are consistent.

There are two things to note when making use of User Functions.

1. When *defining* the function (in NOVA), the syntax must be something like:

   X = USR0 (X, INP_1, INP_2, ...)

   (that is, the "output" value **X** should appear *both* on the Left hand side of the '=' sign and *also* as the first argument in the argument list). This is a bit clumsy, and will disappear when I re-do the expression parser, but you are stuck with it for now.

2. The returned value of the User function is always the *first element of the output vector X* (or the single value **X** if it is not dimensioned). It is **NOT** the "value" assigned to USRn in the case where it is written as a FORTRAN Function subprogram - this value will be *totally ignored* by NOVA (it is safest to *always* write User functions as subroutines and then you won't get confused).

User functions have (or rather, can have) access to any of the variables / constants defined to NOVA (not just the ones passed explicitly as arguments) by passing the argument **$TABLES** to the routine and using system supplied routines to access variables / spectra. See the description at the end of the NOVA OPSEQ manual for details.

Conditions

A condition in NOVA is simply a special kind of INTEGER*4 variable, one that has associated with it a *software scaler.* Every time the expression defining the condition is evaluated, the software scaler is incremented (giving you a count of how often you "tested" the condition) and a second software scaler is incremented if the value of the expression was non-zero (.TRUE.). Thus the ratio of these numbers tells you what fraction of your events passed this particular test. These two values are given when you **LIST** a particular condition. Note that this is all automatic - it is not necessary for the user to explicitly define scalers and include them in the Operation Sequence (as was the case, for example, in DACS).

Conditions are most often used as part of the *spectrum condition mask* (described below) to decide whether a given spectrum should be incremented or not.

As well, there are *implicit conditions* in NOVA associated with every IF statement in the OPSEQ. Every time the IF statement is executed, the a software scaler will be incremented, and a second scaler will be incremented if the expression if the IF statement (either a single variable or a logical expression) is TRUE. The ratio then tells you how often your program "passed" the IF statement.

These software scalers can be viewed with the LIST/IF statement (which lists only the IF statements in the OPSEQ), or the LIST/OPSEQ/FULL command, which lists the whole thing.

Variables in NOVA are only calculated *once per event.* Thus, relying on the software scaler associated with a condition can sometimes be confusing. Consider the following example.

```
A1:   IF (C1) GOTO L1
      .
      IF (C2) GOTO L2
      .
L1:   CONTINUE
      .
A2:   IF (C1) GOTO L2
      .
L2:   CONTINUE
```

Since conditions are also variables, the condition **C1** will be calculated the first time it is required - in the statement **A1**. Thus the counts for **C1** refer to line **A1**, not do not reflect anything about the execution of the IF statement in line **A2**. The implicit counters (LIST/IF) for line **A2** will be correct, however.

Groups

An important part of NOVA is the concept of "groups". A *group* in NOVA is simply a (named) collection of objects of the same type which, in most cases, can be treated as a single object of that same type.

For example, one could define a group of spectra which are all to be incremented at a certain point in the OPSEQ (e.g. within a certain IF block). Instead of including separate *INCR* statements for each spectrum, one could instead write

INCR specgroup

to increment all of these spectra. As another example, one could collect parameters describing a certain set of running conditions (e.g. runs at different energies) into a group of variables, and *LOAD* the appropriate set of parameters whenever the running conditions change. (You *could*, except that selective load of this type is not implemented in the current version of NOVA - I'm working on it).

A group is defined with the statement

ADDGROUP {/TYPE = type} groupname member1 member2 ...

which defines the elements *member1, member2, ...*as belonging to the group *groupname.* If the group doesn't already exist, it will be created - in this case, the *type* of the group (SPECTRUM, VARIABLE or CONDITION) must also be specified.

Items are deleted from an existing group with the statement

SUBGROUP groupname member1 member2 ...

Groups of *conditions* are often used in **IF** statements in the OPSEQ, as in

IF (condgroup) GOTO L1

Conditions in *condgroup* are evaluated (if not already done so for this event) in order until the *first FALSE condition is encountered*, in which case execution continues with the next statement in the OPSEQ (i.e. the GOTO will not be executed). Only if *all* of the conditions in the group are TRUE will the GOTO branch be executed. Thus, a condition group is sort of like the logical AND of all conditions in the group - evaluation of the first FALSE condition terminates the process and the entire group is treated as if it were FALSE (note that further conditions in the group will not be evaluated in this case).

Several groups are predefined by NOVA. In particular, the group **$ALLSPECTRA** contains all spectra which have been defined, and **$ALLCONDS** contains all conditions that have been defined. Examples of their use are given below.

Spectra

The "ultimate output" of NOVA is (usually) a bunch of spectra of certain variable (either **$RAW** data or user calculated variables). This need not be the case (you might only want the values of software scalers for conditions, for example), but most experiments generate at least a few spectra.

Again, the aim in designing NOVA was to allow *anything* about a spectrum to be re-defined "on the fly" (one of the most common criticisms I heard about existing analysis systems was the inability to change the definition of a spectrum).

A spectrum in *incremented* by including an *INCR* statement in the operation sequence. This is discussed in more detail in the NOVA OPSEQ manual.

A spectrum is *defined* with the *DEF1D* or *DEF2D* statement. To define a 1-D spectrum:

DEF1D /SIZE=n /LOW=xlow /HIGH=xhigh /DATA=xdata /CONDITION=(list) specname

To define a 2-D spectrum

DEF2D /XSIZE=n /YSIZE=n /XLOW=xlow /XHIGH=xhigh /YLOW=ylow /YHIGH=yhigh /XDATA=xdata /YDATA=ydata /CONDITION=(list) specname

Things in UPPER CASE must be entered exactly as they appear (although the case is not important - either upper or lower case is OK). Also, the shortest unique abbreviation is recognized (**/S** is OK for **/SIZE**, example). Things in lower case are user variables / constants defining the values you want (e.g. **/XDAT=FOC_PLAN_P**, **/XMIN=-157.21**, **/XMAX=FPL_MAXIMUM)**.

Not all parameters must be entered at once, and (except for the size - I'm working on that too) they can be changed at any time. The only rule is that the *first definition of a spectrum must include the number of channels* (both X and Y for a 2D spectrum). If the limits (**/XMIN, /XMAX)** are not entered, they default to (0.0, n-1). They can, of course, be changed later at any time.

The order of parameters in the definition is not important, *except that the spectrum name must come last!*. Parameters are scanned from left to right, but are only "executed" when the spectrum name is encountered. This makes it easy to define several spectra with the same parameters. For example:

DEF1D /SIZE=100/XLOW=-100/XHIGH=300/XDATA=X SX /XDATA=Y SY

defines spectra SX and SY, both to be 100 channels long, running from -100 to 300. The data histogrammed is *X* for spectrum SX and *Y* for spectrum SY. This order is a little different than what some might be used to, however. The command:

DEF1D SX /SIZE=100

doesn't work!

Note that the limits can be either variables or constants (in some instances, it might be nice to change the limits of a whole bunch of spectra at once by changing just one number - you do this by defining all of the spectra with limits which are variables - E_PION_LOW to E_PION_HI for example). The number of channels (**n**) must be an INTEGER constant, however.

The data is histogrammed in *floating point* (if the **/DATA** variable or any of the limits are not REAL, they will be automatically converted) and negative values are allowed (i.e. you can histogram *real physics variables* - none of this "things must start at channel 0 and let the poor old user figure out what it really corresponds to" stuff).

The *condition list* is a list of conditions (see above) *all of which must be TRUE before the spectrum will be incremented!*

What happens is the following. When NOVA tries to increment a spectrum, it first checks all of the conditions in the condition list for this spectrum (calculating them if necessary). If any of them are FALSE, the spectrum will not be incremented. Note that this test fails when the first FALSE condition is encountered - there is therefore no guarantee that all of the conditions in the condition list will ever be calculated - if the first one is FALSE, none of the others will even be calculated (and hence their software scalers won't be updated - so be careful)! Only if all of the conditions in the mask are TRUE is the spectrum calculated. Each spectrum individually also contains a software scaler indicating how often an attempt was made to increment it and how often this attempt succeeded (i.e. how often all of the conditions in the mask were all TRUE).

Note that each spectrum effectively is governed by *multiple conditions* (not just one, as, for example, in LISA). The total number of different conditions which can appear in the condition lists for *all spectra* is currently limited to 256 - this number could be increased in a future release of NOVA if the demand is present.

Note also that whether or not a spectrum gets incremented is also controlled by the OPSEQ. See the discussion at the end of the OPSEQ document for possible pitfalls in this "double logic" scheme and ways to avoid them.

The empty condition list (**/CONDITION=(0)**) means that there are no conditions on the spectrum (it will always be incremented). This is the default.

Sample Session

This section contains a very simple example of a NOVA session. The assumptions are:

1. You are collecting data from an ONLINE source (VDACS device GRA1:).
2. You have two events (event type 1 = scalers, event type 2 = a single ADC). Your TWOTRAN program to do this is called ADC.EXE
3. You want a spectrum of the raw ADC, and also a spectrum of (ADC - pedestal) for those events where the ADC actually converted and had real data in it.

Once you understand this example, you can look in the OPSEQ manual, which contains somewhat more realistic examples for more complex environments.

Things shown in **underlined bold** text are commands entered by the user. Computer responses / prompts are shown in *italics*. Comments are given in normal type.

First of all, let's set up the Acquisition (VDACS) part. This is covered in greater detail in the relevant VDACS manuals, so will not be discussed in great depth here.

$ **SERVER**
PEWG> **ACQ/BOOT** Initialize the Starburst
PEWG> **ACQ/RUN=ADC/OUT=GRA1:** Load your TWOTRAN program into the Starburst

PEWG> **ACQ/BEG** Start Collecting data
PEWG> **EXIT**

Your TWOTRAN program should now be running and collecting data. Now let's set up a NOVA to analyse it. Note that it is not necessary to have NOVA running to start collecting data (although you might want to - if NOVA is started first it will just sit there waiting for the first event to come along). NOVA is independent of data acquisition and vice versa.

$ **NOVASTART ADC** Start up a NOVA subprocess (assuming you don't have one going already). The VMS process name will be NOVA_ADC.
$ **NOVA** Run the NOVA command line interpreter.
Nova tables are empty The system is simply informing you that you haven't entered anything yet.
Nova> **ADC = $RAW(0)** Tell NOVA that you want to call the first data word in the event ADC
Nova> **1** Insert lines into the OPSEQ. Note that the Nova> prompt goes away to indicate that you are in OPSEQ Insert mode.
0: $BEG_OPSEQ: CONTINUE Nova lists the previous line in the OPSEQ (you are inserting at line number 1 and the preceding line is number 0).

```
IF ($EVENTTYPE == 1) THEN
        EVAL RATES
ENDIF
IF ($EVENTTYPE == 2) THEN
        INCR SRAW
        IF (ABOVE_PED) THEN
                INCR SGATED
        ENDIF
ENDIF
```

^Z CTRL-Z is used to exit OPSEQ insert mode.
Nova> **DEF1D/XSIZE=1000/XDATA=ADC SRAW**
Nova> **DEF1D/XSIZE=1000/XDATA=ENERGY SGATED**
Nova> **ENERGY = ADC - PEDESTAL**
Nova> **CONDITION ABOVE_PED**
Nova> **ABOVE_PED = ADC > PEDESTAL**
Nova> **PEDESTAL = 50**
(See the NOVA OPSEQ manual for the definitions required to handle scalers - Event Type 1).
Nova> **IMODE ONLINE** Tell NOVA where the input is coming from (VDACS device GRA1: in this example).
Nova> **OPEN GRA1:** GRA1 is not necessary here as it is the default.
Nova> **EA** Begin analyzing events.

This is a pretty simple example (and maybe confusing if you haven't read the rest of the manuals). If any of you "unwashed users" would care to create a more instructive one, please feel free to do so.