TRIUMF	UNIVERSITY OF ALBERTA EDMONTON ALBERTA	
	Date 29 December 1995	*File No* TRI-DNA-96-2
Author P. W. Green		*Page* 1
Subject NOVA (Version 2.0) Reference Manual		

NOVA Command Summary

The following contains an alphabetical list of the commands available under version 2.0 of NOVA. Unless explicitly noted otherwise, all commands function identically under Unix and VMS.

In this description, the term *item* refers to anything that the user might define (variable, spectrum, condition, OPSEQ label, ...).

NOVA command lines are as "free-format" as possible (I hate systems where parameters must be separated by EXACTLY one space, or only commas are allowed — especially if the rules are not consistent for all commands). Parameters in NOVA commands are separated with commas and/or one or more blanks or tabs. Qualifiers (which begin with the / character) need not (although may be) separated (from each other or from other words in the command) with blanks or tabs.

Most (not all) commands support *wild cards* in user names. The standard VMS convention applies — the character '%' matches *exactly* one character in the name and the character '*' matches *any number* (including zero) of characters in the name.

In most cases, command lines are "executed" in strict left to right order. Thus, qualifiers should *precede* the arguments to which they apply. For example, to obtain a "full" listing of a variable, you should enter

LIST/FULL variable

rather than

LIST variable/FULL

However, qualifiers at the end of a command line are effectively moved to the beginning, allowing for convenient use of command line recall and editing capabilities[a] (this is described in more detail in the Introduction to NOVA).

All commands (and qualifiers) may be abbreviated to their shortest *unique* form. Thus, **DEL, DELE, ...** are all legal substitutes for the **DELETE** command.

VERY IMPORTANT!

It is *guaranteed* that in this and all subsequent versions of the system, all commands will be unique within the first 4 characters (i.e. a 4 character abbreviation will never cause confusion). Especially in command files, it is a good idea to always use at least 4 characters since future releases of NOVA might introduce new commands which conflict with command abbreviations shorter than 4 characters.

User Aliases

NOVA commands are intended to be as "non-cryptic" as possible. With a few exceptions (*ZALL* is one that comes to mind) it is (I think) fairly obvious from the command word only what the command is supposed to do. This, plus the fact that abbreviations to commands must be unique, means that sometimes more typing is required than in other systems. As one particular example, many systems use the command word **D** to mean display - this is nice and short and therefore

[a] This is new - previous version of NOVA strictly enforced the “qualifiers must precede arguments” rule.

appropriate for something that you commonly use. This won't work in NOVA, because D could mean not only *DSP* but also *DUMP, DELETE, DEF1D, DEF2D, ...*

To allow the user to customize the system to his liking as much as possible, NOVA allows *User defined Aliases.* Any legal name (12 or fewer characters with the first one alphabetic) can be defined to be equivalent to any character string in a command. The only restrictions on the use of User defined aliases are:

1. The command can be used only at the *beginning* of a command line. That is, when the system receives a command line to be executed, it looks at the *first word only* on the line to see if it is an alias. If it is, this word is replaced with the appropriate string, and the command line is then processed normally.

2. This replacement process is *not recursive* (that is, if the first word is replaced with a user command string, the process is not repeated). Thus, aliases may not be defined in terms of other aliases.

3. The total length of any command line in the system (after all of this expansion stuff) is limited to 255 characters.

Thus, for example, if you commonly use the command DSP/PAGE=4 to plot 4 spectra on one screen, you could define an alias called D4 to do this

D4 := DSP/PAGE=4

and then say

D4 S1 S2 S3 S4

which will be expanded to

DSP/PAGE=4 S1 S2 S3 S4

Or, if you just like to use D for DSP, you could define

D := DSP

The syntax for defining an alias is similar to that used for defining a symbol in VMS:

command := replacement_string

The operator ':=' rather than just '=' tells NOVA that this is the definition of an alias and not a variable (note, though, that ':==', which is commonly used in VMS, does NOT work — only a single '=' sign is allowed).

Note that aliases are stored in the NOVA tables along with everything else, so they are destroyed with a LOAD command (and also saved by the DUMP command). In addition, since they are stored in the (common) NOVA tables, everyone who maps to this NOVA gets all of the defined aliases, so you should make sure that everyone agrees on which aliases should be defined. If you want to make use of aliases, it is probably best to keep them all in a command file, and then you can restore all of your definitions after a load with *@file.* You can use the LIST/ALIAS command to see what aliases are currently defined.

If replacement_string is emtpy, the user alias is removed. Thus, to remove the definition of D4, simply enter the command line

D4 :=

General

- *!Comment*

Any command line which begins with an exclamation point is treated as a Comment, and is ignored by NOVA. Trailing comments (i.e. a '!' anywhere but the first character) are not supported.

- *CTRL-C*

Most commands can be terminated by typing CTRL-C (*NOT* CTRL-Y[b] — that terminates your NOVA session). For example, a lengthy LIST command can be aborted by typing CTRL-C, and you will be returned to the NOVA Command Line Interpreter.

- *Command Editing / History*

Any command line may be edited (using the cursor keys to position the cursor, as with VMS DCL) before pressing RETURN to execute the command. As well, any of the most recent 20 commands may be recalled, edited (if desired) and then re-executed by just pressing RETURN. Press PF1 to see the commands which may be recalled, PF2 for HELP and PF4 to invoke a simple desk calculator[c].

The command history is now saved between NOVA sessions (that is if you exit NOVA to DCL and then re-enter NOVA, the most recent 20 commands will still be availble using the up arrow key).

The command editing facility is not available if you have SET/INPUT=REMOTE. This command makes it easier (and <u>much</u> faster) to interact with NOVA over a slow data link (e.g. modem or very slow network connection), but you give up the command editing facility (and also command recall).

A recalled command is added to the command history list only if it is changed (this is to prevent multiple recall of the same command from overwriting your entire command history). If you want an unmodified command to be added to the command history, you can "fake it" by adding a space at the end of the command.

- *@filename*

Begin reading commands from the file *filename.* @-files may be nested up to 10 deep. An @-file is terminated (and hence further command lines will be read from the next higher level) by the physical end of file. If *filename* does not contain an extension, .NCM (for Nova CoMand file) is assumed.

- *NOVA_STARTUP*

When you first type NOVA, the system searches for a file named *NOVA_STARTUP.NCM* in your default directory (or you may use the VMS logical name NOVA_STARTUP to redirect this assignment to any other filename), and if it finds it, executes it as an @-file before your session begins. You could use this, for example, to set your operating environment (e.g. you might want LONG error messages rather than the default SHORT, or you might want to turn on CONFIRMation of all changes). This is most useful for (and in fact should probably ONLY contain) those things which are <u>not</u> saved in the NOVA tables (primarily parameters in the SET command). Otherwise, someone else mapping to your NOVA subprocess (using the DCL NOVAMAP command) and then starting the NOVA program would change definitions for everyone else who is currently mapped to the same subprocess.

- *multiple commands*

Multiple commands can be entered on one input line by separating them with semicolons. This is most useful when commands are passed to NOVA on the command line (perhaps from a command file), although it is available also when NOVA is used in normal interactive mode (it might be useful, for example, to be able to recall multiple commands with the command recall facility described above).

---

[b] In Unix, CTRL-Y does nothing special.

[c] Thanks to Corrie Kost and his group for this facility.

- *n command*

Any command line which begins with a number is assumed to be a line to be inserted into the OPSEQ as line number *n*. If *command* is not present on the line (i.e. just a number is entered on the command line), NOVA enters "OPSEQ insert" mode. Multiple lines may then be entered into the OPSEQ. Use **END, EOF, EXIT** or **QUIT** to return to normal "command" mode. See the comments on the Operation Sequence below for a more complete description).

- *variable = expression*

This defines the formula for calculating a variable. *Expression* can be any legal FORTRAN arithmetic or logical expression (including NOVA function calls and indexed variables). *Variable* must be one of:

i) a simple variable (no subscript)

ii) a subscripted variable which has been <u>previously </u> declared as a PARAMETER. In this case, *expression* may *only* be a simple constant.

iii) the name of a dimensioned variable *with no subscript.* In this case, *expression* may *only* be a call to a USRn function, and the first argument of the user function *must* be the same vector (e.g. **X = USR0 (X, ...)**). (I know this is clumsy, but you are stuck with it — sorry).

Examples:

X = 143.7
LOGVAR = ANGLE >= ANGLO & ANGLE <= ANGHI
Y=(X1-X2)/(X1+X2) + COS(THETA)*LOG(X12)**
USERDATA=USR1(USERDATA,$RAW,$TABLE)

<u>Alphabetical List of NOVA Commands</u>

In the following descriptions, things enclosed in {braces} are optional (in fact for many commands, most of the parameters are optional — the {braces} convention is used in this document only when it is not obvious whether a parameter is optional or not). Things shown here in UPPER CASE must be entered exactly as shown (although the case is not important when you are actually entering the command, and <u>unique</u> abbreviations are always allowed), things shown in lower case are typically user items and should be replaced with the appropriate user-defined values.

- *ADDGROUP {/TYPE=type} groupname member {member ...}*

Add the items *member* ... to the group *groupname.* If groupname does not exist, it will be created - in this case the type of the group <u>must</u> be specified with the */TYPE* qualifier. Note that the first item in the command is the *groupname* - subsequent items are the *members* of the group.

Wild cards are <u>not</u> supported for the ADDGROUP command.

The command SUBGROUP is used to remove items from a group.

Qualifiers:

/TYPE=type	Defines the type of the group. This qualifier is *required* when the group is first created. Legal group types are **CONDITION, VARIABLE** and **SPECTRUM**.

- *ANALYSE*

Begin analysis. NOVA first checks to ensure that an input stream has been opened (see the OPEN command). OPEN in turn requires than an IMODE command be executed first.

NOVA then checks for a "non-executable" OPSEQ (e.g. unpaired IF - ENDIF construction). If any exist, analysis is not allowed.

NOVA then checks for any undefined entries (i.e. things which have been referenced somewhere, but NOVA doesn't yet know how to calculate them — an example would be defining X=Y+1, but forgetting to define what Y is). If any exist, the system asks if you really want to begin analysis. As far as I know, executing an OPSEQ containing undefined variables will not make the system crash, but the results will almost certainly be wrong. It is safest to proceed from this warning message only if you are sure that the undefined variables are in a part of the OPSEQ which will never be executed (or that you don't care about yet).

EA (Enable Analysis) is a synonym for ANALYSE (as is ANALYZE, for those of us who can't figure out if we are British or American).

- *ANALYZE*

A synonym for ANALYSE.

- *BREAKIN*

As discussed below (under the LOCK command), it is possible for the owner of a NOVA sub-process to prevent modification by anyone else (although the default is that any terminal / process which can access the tables can change them[d]). This locking facility can be useful if you are worried about an unauthorized (or careless) user making changes to things, but it can also be a bother if you "know what you are doing" but find yourself at a place remote from the terminal which has locked the subprocess (e.g. you are on the MRS platform and the NOVA that you want to talk to has been locked by a terminal in the counting room).

If you want a different terminal to be able to change things, the "owner" terminal (usually) must first execute an UNLOCK command, and then the new terminal can issue the LOCK command to gain exclusive write access to the tables. The current "owner" of a particular NOVA subprocess is displayed on the STATus page.

The BREAKIN command allows you to force your way into gaining write access to the NOVA tables. You then become the owner of the subprocess and can change definitions (it is just like the LOCK command, except that the other guy is not required to issue an UNLOCK first).

This is potentially a very dangerous (and confusing) thing, since there is no indication given to the other process that this has happened (it might not even be your NOVA process that you are breaking into — please be *VERY CAREFUL* when using this command — try to avoid it if at all possible). In order to slow you down a little bit (and maybe make you think carefully about what you are doing), the system requires that you enter a Breakin Password (the "Password" is simply the BREAKIN command word again — no short forms allowed and it must be in UPPER CASE). The intent is not so much to prevent you from using this facility (I am assuming that people will use it responsibly, and it isn't much good having it if you can't use it) as to make you think about it before you do. A real password may have to be implemented if this facility is abused.

Once you have done this, you will remain the sole owner of this NOVA process until you UNLOCK it (or until someone else does a BREAKIN to steal it away from you without your knowledge— see what I mean about it being confusing and dangerous).

- *CENTROID*

The CENTROID command closely mimics the DSP command described later — in particular it accepts (most of) the same qualifiers. The difference is that rather than displaying a spectrum, it calculates statistics for all of part of it. In

[d] This is different from earlier versions, where only a single process was allowed write access to the NOVA tables.

particular, it calculates the Sum, Centroid, FWHM (Full Width at Half maximum = 2.35 σ) and Standard Error of the Mean (SEOM). This information is either displayed on the screen or written into a file (appended to the file if it exists already).

Qualifiers:

Most of the qualifiers which are legal for the DSP command are also recognized for the CENTROID command (e.g. /XMIN, /XMAX specify the region of the spectrum to use in calculating the Sum & centroid). The /ROW and /COLUMN qualifiers are particularly useful for specifying a single row or column of a 2D spectrum. One qualifier unique to the CENTROID command is:

/FILE Causes the information to be written to the disk file CENTROIDS.DAT rather than to the screen. If this file exists already, the information is appended to the end of it (allowing, for example, a history of centroids for different runs to be maintained in a single file).

- *CLOSE*

This command closes the data stream opened with the OPEN command. If the input stream is closed, then a new input stream must be opened before analysis can be resumed. The NOVA subprocess must be in a *paused* condition before the CLOSE command will be accepted.

Qualifiers:

/INPUT Closes the input data stream. This is the default if no qualifier is entered (i.e. CLOSE is equivalent to CLOSE/INPUT).

/OUTPUT Closes the output data stream. All OUTPUT statements in the OPSEQ will be ignored until another output stream is opened. This is the easiest way to disable OUTPUT from the NOVA OPSEQ.

- *CONDITION name {name ...}*

Declare one or more variables to be of type *condition* (i.e. an integer value with an associated software scaler). Each *name* must be a simple variable (conditions may not be dimensioned). These may be either new variables (i.e. not yet defined) or already existing. If they exist already, their definition (i.e. how they are calculated) will not be changed. The only difference will be that their type will be forced to INTEGER and there will now be a software scaler associated with them.

- *CURSOR*

The CURSOR command turns on the display cross-hairs. They can be moved with the arrow keys on the terminal (or the mouse if you have an Atari or DECWINDOWS terminal). This command can be used if multiple plots are on the page (i.e. DSP/PAGE=n), but any commands which cause a spectrum to be redrawn (such as Expand) will redraw only a single spectrum — the one in which the cursor is currently positioned. Note that the Markers (Left and Right) must both be defined within the same spectrum.

You interact with this command by moving the cursor (either with the mouse or the arrow keys) and/or typing single character commands. You need not (in fact, MUST not) enter a Carriage Return - just the single character key is sufficient.

The top line of the screen contains the X (and Y for 2D) coordinates of the Left marker on the left hand side of the line and of the Right marker on the right hand side of the line. The second line of the screen displays different things for different functions.

For commands which define a channel (e.g. **L**, **R** or **C**), the crosshairs will normally be re-positioned to lie at the centre of the selected channel. If you don’t like this behaviour, it can be turned off with the SET /SNAP command.

The following single character commands are implemented within the CURSOR command.

n Entering a single digit (1 - 9) displays as a full screen one of the first 9 spectra which were shown on a multi-plot screen with the last DSP command. Use the All command to restore the screen to its oritinal state.

A Redraw **A**ll of the spectra which were on the screen when the CURSOR command began.

Space or **C** Display the **C**ounts in a particular channel (the X and Y data values associated with the centre of the channel are also given). The current position of the crosshairs selects the channel.

D **D**raw the spectrum again (but with updated data — if analysis is going on in parallel, you will get a plot with the current data). If there are multiple plots on the screen, only the one pointed to by the cross-hairs will be drawn. (Contrast this with the **A**ll command, which redraws everything). This is similar to the 'n' command above, but gives you access to more than the first 9 spectra.

E **E**xpand the spectrum between the Left and Right markers. Note that the Left and Right markers must both be defined within the same spectrum.

F **F**ull display of the spectrum (expand so that all channels are displayed — including Underflow and Overflow channels).

G **G**aussian fit between Left and Right markers (this has not yet been implemented).

L **L**eft marker for a region. Usually, the Left button of the Atari mouse is set to generate an **L** when you click it. If it is not, the ST640 setup screen — invoked by the Help button — allows you to set this. Note that the Left and Right markers must both be defined within the same spectrum.

Q **Q**uit (terminate the CURSOR command and return to the NOVA CLI). This is the only way to escape from the cursor command.

R **R**ight marker for a region (usually the Right button of the Atari mouse is set to generate an R when you click it). If not, it can be set in a manner similar to the Left button. Note that the Left and right markers must both be defined within the same spectrum.

S **S**um the region between the L and R markers (gives the centroid and FWHM as well). The region being summed is displayed as two dotted lines (for a 1D spectrum) or a dotted box (for a 2D spectrum). Note that the Left and Right markers must both be defined within the same spectrum.

W Draw the **W**indow (L and R markers). Note that if multiple spectra are on the page, there is only one Window defined (not a separate one for each spectrum) and both the L and R markers must be within the same spectrum for the window definition to be valid.

- *DCL {command}*

If *command* is specified, a subprocess is spawned and the single DCL command is executed. Control then returns to the NOVA CLI. If no *command* is specified, then a subprocess is spawned running VMS DCL (or the Unix shell). Input is read from the terminal and executed as DCL (not NOVA) commands. The terminal prompt is changed to NOVA_DCL> to remind you that you are talking to DCL from within NOVA. Use the DCL LOGOUT (Unix exit) command to terminate this subprocess and return to NOVA.

Note that commands are executed by spawning a subprocess, which may not be possible if you have too many subprocesses active already (operating system quotas apply). For example, if you have started a couple of NOVA subprocess and also an NSCALERS subprocess to monitor scalers, you may be unable to execute DCL commands from within NOVA due to subprocess quota restrictions. You can get around this by terminating some of your processes (and perhaps restarting them from another terminal / parent process).

The DCL command may be combined with the User Alias utility to create an interface which is similar to VMS DCL. For example, to generate a directory listing from within NOVA, define the User Alias

DIR := DCL DIRECTORY

and the (NOVA) command DIR behaves just like you were talking to DCL (although maybe quite a bit slower)— It spawns a subprocess, executes the command DIRECTORY and then returns to NOVA.

- *DEF1D /qualifiers ... spec1 {... spec2}*

Define (or change the parameters of) one or more 1-D spectra. Note that the spectrum name comes *last* (i.e. the command line is scanned left-to-right, but is "executed" only when the spectrum name is encountered - all qualifiers before *spec1* refer to spec1 - all qualifiers before *spec2* (including those before spec1 if there are no conflicts) refer to spec2 (See however, the discussion on page 1 regarding the placing of command qualifiers). Not all qualifiers are required in a given DEF1D command (they can be entered later), but *if the spectrum must be created (doesn't exist already) the /SIZE parameter must be given*. A spectrum is "undefined" until both the SIZE and DATA parameters have been specified — there are reasonable defaults supplied for all other parameters.

Wild cards are supported (you can say DEF1D /XLOW=-500 SR* to change the lower limit of all spectra beginning with SR).

Any parameter of a spectrum *except the size* can be changed by a subsequent *DEF1D* command at any time.

Qualifiers:

/CHART

Defines the spectrum to be of type strip-chart. This is like a SEQ spectrum (see below), except that rather than “wrapping around” at the end of the spectrum, the spectrum is scrolled left by 1/2 the size.

/CONDITION=(list)

Defines a list of 1 or more conditions in the condition list for this spectrum. *All* of these conditions must evaluate to TRUE for the spectrum to be incremented. An empty condition (/COND=0)list means that the spectrum will *always* be incremented. This is the default.

If only a single condition is given, the parentheses may be omitted.

Conditions in this list will be *added* to the existing condition list for the spectrum (you don't have to re-type what is already there to add another condition). If the condition name is preceded with a '-' sign, this condition will be *removed* from the condition list. The special condition name **0** (zero) removes all conditions from the list.

Thus, to remove condition COND1 and include COND2 and COND3, you would specify:

/CONDITION=(-COND1, COND2, COND3)

To define the mask as COND1 only (regardless of what was there before) specify:

/CONDITION=(0, COND1)

/DATA=var (or /XDATA)

Defines the data value to be histogrammed. Values with a subscript are not allowed. If you want to histogram $RAW(10) (for example), you must first define an auxiliary variable (e.g. R10 = $RAW(10)) and then use R10 as the /DATA variable.

/LOW=var (or /XLOW) Defines the value of XDATA associated with lower edge of channel 0 of the spectrum. Defaults to 0. The value may be either a constant or the name of a variable.

/HIGH=var (or /XHIGH) Defines the value of XDATA associated with the upper edge of channel (SIZE-1) of the spectrum. Defaults to SIZE. (Actually, XHIGH is the first value which is NOT stored in the spectrum, or at least the first value which is stored in the overflow channel. The top of the last bin in the spectrum is XHIGH-ϵ). The value may be either a constant or the name of a variable. For example, if you specify /XLOW=0, /XHIGH=100 and /XSIZE=100, each channel will be 1.0 data units wide. The first channel covers the range (0.0,0.999...) and the last channel covers the range (99.0, 99.999...). The 'bottom' of the overflow channel is 100.

Note that there is *no requirement* that (XHIGH - XLOW) be an even multiple of XSIZE (although it usually makes it easier to interpret channel contents if this is the case). The "Bin Size" of the spectrum is (XHIGH - XLOW)/XSIZE as a *floating point number*.

/NORMAL Defines the spectrum to be a "normal" 1D spectrum (see /CHART./PTOS and /SEQ). This is the default.

/PTOS Defines the spectrum to be a"Parallel-to-Serial" spectrum. The data value being histogrammed must be of type INTEGER. Channel 'n' of the histogram will be incremented if bit 'n' of the data value is set (bit 0 is the least significant bit). Thus, this type of spectrum can be incremented several times for each event (once for each bit which is set in the data value). A /SIZE parameter greater than 32 is meaningless (although allowed) as there are only 32 bits in an INTEGER*4 variable. The Underflow channel will be incremented if the value is zero (no bits set). The Overflow channel will never be incremented.

/SEQ Defines the spectrum to be of type sequential. A sequential spectrum differs in that, when it is "incremented", the value of the variable being histogrammed is stored in the next channel of the spectrum. Thus, a sequential histogram gives you a time ordered history of the value of a variable, rather than a histogram of how often a variable had a certain value. It might be used, for example, to maintain a history of some parameter (such as beam polarization) as a function of time. A spectrum of type /CHART is similar except that is scrolls by 1/2 the size rather than "wraps around" when the last channel is reached.

/SIZE=n (or /XSIZE) Define the number of channels in the spectrum (>0, INTEGER constant). This qualifier is *required* when the spectrum is first defined.

/STORE Defines the specgrum to be of type 'store vector'. When it is "incremented", the value of the X data variable (which should be a dimensined vector) is simply copied into sequential channels of the spectrum.

Examples:

DEF1D /SIZE=100 SX
DEF1D /XDATA=X /XLOW=-100 /XHIGH=200 SX
DEF1D /CONDITION=0 SX

- *DEF2D /... spec1 {/... spec2}*

Define one or more 2D spectra. See the comments for *DEF1D* above - the *DEF2D* command is similar. Qualifiers must include either an X or Y in them (/SIZE is no good - it must be either /XSIZE or /YSIZE). When the spectrum is first defined, *both /XSIZE and /YSIZE must be given.* Note that the maximum size for a 2D spectrum is approximately 256 x 256 channels (this limit is actually imposed by the DSP command — larger 2D spectra can be defined and incremented OK but they can't be displayed properly by NOVA. You have to use PLOTDATA or some other graphics package to display them). The system will issue a warning if you try to define a spectrum that is too big, but it will go ahead and do it for you anyway. If you want to undo it, you must first DELETE the spectrum and then re-define it, as NOVA can't change the number of channels in a spectrum

Qualifiers:

/CONDITION=(list)

/INDEX Defines the spectrum to be of type INDEX. An INDEX spectrum (although declared as a conventional 2D spectrum) is really a series of 1D spectra. The XDATA variable should be a dimensioned vector — the YDATA variable is irrelevant and should not be specified.

The first time the spectrum is incremented, the value of XDATA(0) is used to increment a 1D spectrum stored in the first row. The second time it is incremented (for the next event), the value of XDATA(1) is used to increment the second row as a 1D spectrum and so on. After the N'th (= YSIZE) row, the index wraps around and the next time row 0 is incremented again. Thus, an Index spectrum is a convenient way of monitoring a complete data vector with only one INCR call in the OPSEQ. Note, though, that each 1D spectrum is incremented only every N'th event (where N is YSIZE).

/NORMAL

/PTOS If specified, both X and Y axes are assumed PTOS-type variables — there is no provision for one PTOS and one NORMAL coordinate. Note that the corresponding /XDATA, /YDATA variables must both be of type INTEGER.

/XDATA=var

/XLOW=var

/XHIGH=var

/XSIZE=nx

/YDATA=var

/YLOW=var

/YHIGH=var

/YSIZE=ny

Examples:

DEF2D /XSIZE=50/YSIZE=50/CONDITION=C1 SXY
DEF2D /XDATA=X /YDATA=Y /CONDITION=(-C1, C2) SXY

- *DELETE name {name ...}*

Delete one or more items. Entries will be deleted (if possible — that is, if they are not being used somewhere else). If the entries are being used, they will be marked as Undefined but not removed.

Wild cards are supported by the DELETE command.

System variables (i.e. things that NOVA pre-defines for you, such as **$CONDMASK**) are protected. Thus, the command

DELETE *

deletes all user variables but *not* system defined variables.

It is not possible to delete something which is referred to by something else. For example, if you have defined

X=Y+1

you can't delete Y because NOVA recognizes that Y is required for the calculation of X (you could delete Y if you deleted X first). If you try, NOVA will complain that the variable is "in use" and therefore can't be deleted. Its definition (i.e. how its value is calculated) will be removed, however, and it will appear as an Undefined variable (i.e. as if you had just entered the above definition for X but not yet defined Y).

Qualifiers:

/CONFIRM Items to be deleted are listed on the terminal. The user must then enter one of the single letter commands:

Y	Yes - delete this item
N	No - keep this item
A	All - delete all the rest (i.e. just like **Y** for all remaining items in the list)
Q	Quit this command (i.e. just like N for all remaining items in the list)

/ORPHAN Only those items which are "orphans" will be considered as candidates for deletion (an *orphan* is an item which is not referred to by anyone else, and hence could be deleted without changing things). If you want to delete all orphan entries, you must enter '*' for the name.

/VERIFY Names of all deleted items will be echoed on the console.

Examples:

DELETE X1, Y1, Z*
DELETE/CONFIRM *
DELETE/ORPHAN *

- *DELETE/OPSEQ n1 {n2}*
- *DELETE/OPSEQ ALL*

Delete lines from the OPSEQ. This statement is (somewhat) obsolete now that the full screen editor for the OPSEQ has been implemented, but is still useful in cases where your terminal (or operating system!) does not support a decent screen editor.

If only n1 is given a single line will be deleted (the first line in the OPSEQ is line 1 — line "0" is the (protected) $BEG_OPSEQ:CONTINUE). If both n1 and n2 are given, lines n1 to n2 (inclusive) will be deleted. If n2 is entered as ALL, all lines from n1 (inclusive) the the end of the OPSEQ will be deleted. If only ALL is entered, the entire OPSEQ will be deleted.

The first line (**$BEG_OPSEQ:CONTINUE)** and the last line **($END_OPSEQ:CONTINUE)** of the OPSEQ are protected, and cannot be deleted. Thus a command like

DELETE/OPSEQ 10 999999

would delete all lines from 10 through to the end of the OPSEQ (but not $END_OPSEQ).

Qualifiers:

/CONFIRM

/VERIFY As for the DELETE command above.

- *DIMENSION var(size) {var (size) ...}*

Define one or more variables as *vectors* (things requiring an index). *Size* is the number of elements in the vector - <u>the minimum index is always 0</u> (and hence the maximum index is size-1). The variables must not have been dimensioned before (you can only do this once - I can't change the size of a dimensioned variable) and it must not have appeared before without an index (since this implicitly defines it to be a simple, undimensioned variable, and I can't change that either). Variables can also be dimensioned in type declarations statements (e.g. INTEGER*4 X(40)).

- *DISMOUNT*

This command logically dismounts a magnetic tape (it is the inverse of the MOUNT command). It is required (for example) if you have been analyzing from tape and want to analyze a disk file instead. See the section on tape handling for a more complete description.

- *DSP /... name1 {, /... name2 ...}*

Displays one or more spectra on the terminal. The command string is scanned left to right, but only "executed" when a spectrum name is encountered. Thus, all qualifiers preceding *name1* affect the display of *name1*, all qualifiers preceding *name2* (including those which precede *name1* where there are no conflicts) affect the display of *name2*, etc. (See however, the discussion on page 1 regarding the placing of command qualifiers. In particular, any qualifiers which follow the last spectrum name are effectively transferred to the beginning of the command line and hence will apply to all spectra in the command. This makes it more convenient to recall a command and add qualifiers to the command.)

Each *name* may be either a single spectrum name, or the name of a spectrum *group*. If it is a spectrum group, the effect is as if all members of the spectrum group (modified by any preceding /OFFSET qualifier, if present) were entered instead of the spectrum group name. Wild cards are supported for the DSP command — both individual spectra and spectrum groups are searched to find a match for a name containing wild cards. Spectra which are included in groups may appear more than once if wild cards are used. For example, if a spectrum SX1 is included in a spectrum group SXPOS, then the command DSP SX* will display SX1 twice — once because it is a member of SXPOS (which also satisfies the wild card condition). The spectrum group $ALLSPECTRA is ignored during this wild card match, so the command DSP * displays each spectrum only once.

Multiple spectra may be displayed on one page.

Qualifiers:

/ATARI Draw the spectra using the "fast histogram" option of the Atari ST640 package. This is the default if the terminal type has been set to Atari using the SET command. It is much faster than the /VT640 option, but (of course) requires that you are using an Atari 1040 running ST640. In addition, spectra produced with this option may <u>not</u> be printed on a REMOTE printer using the HARDCOPY command.
This qualifier is "remembered", so once it has been used, all further plots will be drawn using the Atari Fast Histogram package until it is explicitly cancelled with a DSP/VT640 command.

/COLUMN=value Display a column (fixed X data value) of a 2D spectrum as a 1D spectrum. Value is the X <u>data value</u> (not bin number) of the column to be displayed (in fact, any data value contained within the limits will work).

/FULL Normally, the first and last channels of a 1D spectrum (underflow and overflow) or the outer "box" around a 2D spectrum is ignored when displaying the spectrum. The /FULL option forces these channels to be displayed.

/HIGH A synonym for /XHIGH when displaying a 1D spectrum.

/LOW A synonym for /XLOW when displaying a 1D spectrum.

/MAX=n Force the maximum vertical scale to be *n* counts. All channels with counts >= n will be displayed as "full scale". If not entered (or entered as /MAX=0), the scale will be adjusted automatically so that the largest channel contents fit on the plot.

/MIN=n Force the minimum value on the vertical scale to be *n* counts. Normally the minimum value is 0. This option is particularly useful for histograms where the value in a channel can be negative (e.g. a /SEQ spectrum of beam polarization, or spectra which have been incremented using a negative WEIGHT).

/OFFSET=n When the name of a *spectrum group* is encountered in the command line, the effect is as if all of the members of the spectrum group were entered in place of the group name. The /OFFSET qualifier modifies this by starting with member number *n* of the group (the first spectrum is number 0). For example, if *SPECGROUP* is a spectrum group containing 12 spectra, the command

DSP /OFFSET=9 SPECGROUP

would display the last three spectra (numbers 9, 10 and 11) in the group.

/PAGE=n Format the display page so that a maximum of *n* spectra can be drawn on the page. Values are effectively rounded up to the next larger number of the series (1, 2, 4, 6, 9, 12, 16, 20, 25). If this option is not used, the format will be chosen automatically based on the number of spectrum *names* in the command line. Note that (for example), if /PAGE=3 is specified, the display page will be formatted for 4 plots (the next larger number in the above sequence) but only 3 plots will actually be produced.

Numbers larger than 9 produce plots on which very little detail is visible.

/ROW=value Display a row (fixed Y data value) of a 2D spectrum as a 1D spectrum. Value is the Y data value (not bin number) corresponding to the row to be displayed.

/VT640 Draw the plots using the TRIUMF standard graphics driver for a VT640 terminal. This is the default (unless you use the SET TERMINAL command to modify it). It can be used if the terminal you are using is an Atari, but it is much slower than the */ATARI* OPTION. It does, however, permit the HARDCOPY command to produce plots on a REMOTE printer.
Use /VT640 for all terminal types except Atari. For example, if your terminal really is a VT241 (and has been declared as such with the SET /TERM command), then DSP/VT640 will use the VT241 driver. This option really means “don’t use the Atari Fast Histogram display package - use the TRIUMF GPLOT routines.

/XHIGH=value The largest channel displayed in the X direction will correspond to XDATA = value. The synonym /HIGH may be used for a 1D spectrum. If not entered, the default is the last channel of "real data" (i.e. excluding the overflow channel) unless the /FULL qualifier is used.

/XLOW=value The smallest channel displayed in the X direction will be that corresponding to XDATA = value. The synonym /LOW may be used for a 1D spectrum. If not entered, the default is the first channel of "real data" (i.e. excluding the underflow channel) unless the /FULL qualifier is used.

/XPROJ Displays an X projection of a 2D spectrum as a 1D plot. For each X channel, all Y channels between /YLOW and /YHIGH will be summed (i.e. you can produce a projection of part of a spectrum). The underflow and overflow channels will not be included unless the /FULL qualifier is entered.

/YHIGH=value The largest channel displayed in the Y direction will be that corresponding to YDATA = value. If not entered, the default is the last channel of "real data" (i.e. excluding the overflow channel) unless the /FULL qualifier is used.

/YLOW=value The smallest channel displayed in the Y direction will be that corresponding to YDATA = value. If not entered, the default is the first channel of "real data" (i.e. excluding the underflow channel) unless the /FULL qualifier is used.

/YPROJ Displays a Y projection of a 2D spectrum as a 1D plot. For each Y channel, all X channels between /XLOW and /XHIGH will be summed. The underflow and overflow channels will not be included unless the /FULL qualifier is entered.

- *DUMP filename*

Dumps the NOVA tables and/or spectra to a disk file. The LOAD command is used to retrieve the information at a later time.

The *DUMP* command checks for undefined entries, and warns you if any are present before it creates the dump file. You are asked for confirmation before NOVA will generate a DUMP file which is incomplete.

Qualifiers:

/ALL Dump both the tables (definitions) and the spectrum contents to *filename.* This is the default (except for /ASCII).

/ASCII Dump the definitions only (no spectrum contents) as an ASCII (i.e. human-readable) file. This file can be edited using a standard text editor. If no extension is given, the default .NCM (<u>N</u>ova <u>C</u>o<u>M</u>mand file) is assumed.

/BINARY Dump the information in Binary (i.e. computer- readable form). This is the default. If no file extension is given, the default .NBI (<u>N</u>ova <u>BI</u>nary file) is assumed. Dumping and loading Binary files is <u>much</u> faster that ASCII files, and they may also include spectrum contents. Note, though, that such Binary dumps are NOT transportable across different computer platforms (you cannot read a Binary DUMP created under VMS on an Ultrix computer, for example). Only ASCII dumps can be transported between different computer types.

/SPECTRA Dump *only* the <u>contents</u> of the spectra (no definitions). It is up to the user to ensure that, when this file is later LOADed, the table definitions match the ones that were in effect when the spectra were dumped. Use of this option is <u>not recommended</u> until I can think of a way to protect you against table / spectrum incompatibilities.

/TABLES Dump <u>only</u> the tables (i.e. definitions) — contents of histograms are not dumped. This is the default (in fact, all that is allowed) for /ASCII. For Binary files, it results in dump files which are much smaller (and therefore faster to transfer over a network link).

Note that the format of the dump file is controlled by qualifiers, <u>not by the extension on the file name!</u> (I emphasize this only because I have been caught by it a number of times). The command DUMP JUNK.NCM will create a <u>BINARY</u> dump (the default DUMP file type is binary) in a file called JUNK.NCM. The LOAD command is clever enough to figure out the type of the file — the extension is not used - so this causes confusion only for people, not the program.

- *ECHO {string}*

If *string* is present in the command line, then this command will simply type it on the terminal. This might be ueful for inserting comments in a command file to monitor its progress.

If *string* is not given (i.e. just the command word *ECHO* is entered), then all further commands read from command files will be listed on the terminal before they are executed. The command SET ECHO does the same thing. Use the command *NOECHO* (or SET NOECHO) to disable this function.

- *EDIT name*

This command lists the FORTRAN expression which was used to define the variable *name*, and allows you to edit it using the cursor keys on the terminal (similar to the VMS RECALL command, except that you are recalling the definition of a variable rather than a recent command line). Note that only *variables / conditions* can be edited (it is not needed for spectrum definitions — you can just enter the new definition of a parameter in a spectrum definition and it will supersede the old one). You cannot use this facility if you have SET /TERMINAL=REMOTE.

- *EDIT/OPSEQ*

This command invokes a full-screen editor to edit the OPSEQ. The first line ($BEG_OPSEQ: CONTINUE) and the last line ($END_OPSEQ: CONTINUE) will not be included (since you had better not delete them, I just don't give you the option).

Under VMS, the editor works just like EDT (in fact, it *IS* EDT with the exception that no .JOUrnal file is produced). In particular, you should use EXIT to exit the editor and save the changes that you have made in the OPSEQ, and use QUIT to exit with no changes (i.e. abort and leave the OPSEQ as it was).

If you have a favourite editor command file to customize EDT, define the VMS Logical Name EDTINI before you start NOVA.

$ DEFINE EDTINI yourstartupfile

Under Unix, the particular editor that is invoked is specified by the environment variable EDITOR. Before you begin your NOVA session, you should

setenv EDITOR your_favourite_editor

NOVA spawns a subprocess to run the editor, and waits until it returns. The only assumption made by NOVA about the editor is that if you abort (i.e. QUIT to not save changes), it will not create a new file. NOVA uses the modification time of the (internally generated) disk file to decide if the changes made in the editor should be saved or not. If, when the editor returns, the modification date of the file is later than when it was created, the changes will be incorporated into the OPSEQ. The granularity of this time stamp is 1 second, so unless you can type VERY quickly, this should guarantee that only changes that you really want are incorporated (if this causes a problem, please contact me).

- *END*

This is a synonym for the EXIT command.

- *EOF*

This is a synonym for the EXIT command. It is used also (often in command files) to terminate OPSEQ insert Mode.

- *ERASE*

Erases the text, graphics or both screens on your terminal. (Selective erase of only one screen may not be supported for all terminal types / hardware).

QUALIFIERS:

/ALL Erases both the text and the graphics screen. This is the default.

/GRAPHICS Erases only the graphics screen. The text screen is unaffected.

/TEXT Erases only the text screen. The graphics screen is unaffected.

- *ERUN {...}*

The ERUN command is included to allow the NOVAMAIN subprocess to perform some (unspecified) function at the end of a run. It causes NOVAMAIN.EXE to call the (user supplied) subroutine USRERUN, passing to it the remainder of the command line (if any). It is similar to the USREXE command (in fact identical, except that it is intended to be used only at the end of a run).

- *EVAL variable*

The EVAL command evaluates the current value of a variable, using information from other variables already in the NOVA tables. It has the same effect as the EVAL statement in the OSPEQ. As a simple example, if you had entered the definitions

X=1
Z=(X+1)2**
Y=2*Z+3

the command

EVAL Y

would evaluate the expression for Y (using the value X=1) and set Y to 11 (and also print the value of Y on the terminal). As a side effect, the value of Z would also be set to 4 (although this is <u>not</u> printed at the terminal). This facility is very useful for debugging purposes.

A major enhancement of the EVAL command in this version of the system is that the expression that you are evaluating may now refer (either directly or indirectly through intermediate variables) to USRn functions that you have defined. Again this is useful primarily for debugging — you can set up all of the variables that your USR function needs, and then EVAL it and look at the results.

As it goes through the chain of intermediate variables, the EVAL command checks for anything which is undefined. If it finds any, it reports it and terminates the command. The final value reported for the variable which is being EVALed will <u>not</u> be correct in this case.

- *EXIT*

The EXIT command is used primarily to terminate "OPSEQ insert" mode (i.e. stop entering lines into the OPSEQ) and return to normal command mode. The END, EOF and QUIT commands, or the key sequence CTRL-Z[e] are synonyms for EXIT.

If entered from the console in command mode (i.e. when you are not entering lines into the OPSEQ), any of these commands terminates the NOVA program and returns you to VMS DCL.

Qualifiers:

[e] Do NOT use CTRL-Z with Unix.

/SAVE When you exit NOVA "normally" (e.g. with the EXIT command rather than CTRL-Y), the backup file (NOVA_xxxxxx.TBL) is deleted - the system assumes that you are happy with everything that you did and don't want to retain the older version. If you use the /SAVE qualifier, the backup file will be retained. If you wish to keep the file permanently, you must rename it to prevent the system from overwriting it the next time you run NOVA.

- *GCOLL*

Force "Garbage collection" to occur. All items in NOVA are stored in one large table. As things are deleted (either explicitly or internally by NOVA) "holes" will be left in the tables. This command reorganizes the tables to eliminate these holes, and also reports how much of the table is used.

This command is (almost) obsolete, since garbage collection is now automatic after every NOVA command is executed. It does, however, do a (minimal) check on the table structure to ensure that things haven't become corrupted.

- *HARDCOPY*

Produces a hard copy output of the graphics screen (i.e. what you did most recently with the DSP command). Threeodes are supported:

1. If you have SET /PRINTER=REMOTE (or used the /REMOTE qualifier in the HARDCOPY command), the plot will be printed on the VAX printer queue HPLASER[f] (you must have defined this VMS logical name before entering NOVA). Remote printing is *NOT SUPPORTED* if the last display was generated using the /ATARI option (either explicitly in the DSP command or implicitly because of the SET /TERMINAL type). That is, the graphics screen must have been produced using the TRIUMF Graphics package GPLOT.

2. If you have SET /PRINTER=HPLASER or EPSON (or used the /HPLASER or /EPSON qualifier in the HARDCOPY command), the assumption is that the printer is a Local Printer (i.e. attached directly to an Atari 1040). The appropriate printer codes will be sent directly to the printer port of the Atari.

3. If you SET /PRINTER=HPPCL, Hpxxx or PSCRIPT the plot will also be generated on the (remote) printer queue HPLASER. You must generate the display using the XWINDOWS display driver (not DECWINDOWS) to use this.

Printing to a remote printer by default uses the commands:

```
PRINT/PASASALL/QUEUE=HPLASER filename (VMS)
lpr -x -h filename                    (Unix)
```

Where ‘filename’ is the name of an (internally generated) file which contains the picture to be printed. Occasionally printer queues (especially Unix) may be configured so that a different command is needed. In this case you can use the NOVAPRINT environment variable to specify the command to be used to send the job to the printer. In this environment variable, the string ‘%f’ represents the filename. For example, if the “-h” flag doesn’t work on your system for some reason, you could define

```
setenv NOVAPRINT “lpr -x %f”
```

To use the (VMS) command HPRINT to print a plot in PostScript format you would

[f] This was HP$LASER in earlier versions of NOVA. You might have to change a logical name assignment in your LOGIN.COM to use the new system. The easiest way (to remain compatible with other TRIUMF software) is DEFINE HPLASER HP$LASER

DEFINE NOVAPRINT “HPRINT %f W”

Note that (as this variable typically includes spaces and/or special characters) it must be enclosed in double quotes when it is defined.

Qualifiers:

/COLOUR	Forces printing 2D plots in colour (except that colour printing is not yet supported - it is “faked” using differing levels of gray-scale shading).
/EPSON	
/FILE=name	Rather than actually sending the data to a printer, it is saved instead in the named file. This is most useful for PSCRIPT mode - the file generated can (for example) be MAILed to a colleague. You can use this also for HPPCL and Hpxxx printer modes.
/HPLASER	
/REMOTE	This is the default (in fact, all that is allowed) if you have SET/PRINTER=PSCRIPT, HPPCL, HPxxx.

- *HELP {topic {subtopic ...}}*

If no *topic* is given, the system lists the topics available. This is very similar to the VMS HELP command.

- *IMODE mode*

This selects the format (and in some instances the physical source) of the input data expected by the analysis subprocess (i.e. the Input MODE).

NOVA can analyse data written by many different acquisition programs, but it must be told what the data format is. The formats currently recognized are:

CODA The CEBAF On-line Data Acquisition system. The particular source of the data (i.e. Tape/disk or a real-time process) is specified by the file name used when the input stream is OPENed — see the OPEN/INPUT command for details.

DAQ Online data from a DAQ process.

FASCII As the name suggests, this format reads ASCII data from a disk file (it has been used, for example, to get the output from a Monte Carlo code into NOVA). The data in the disk file is assumed to be in Floating Point format and to consist of fixed length events. Data is in free-format (see a discussion of the READ command below), with the exception that each event is assumed to begin on a new line (i.e. data from one event can appear on several lines in the input file, but if there are any “left-over” values on the last line, they are simply discarded).

You must define the following variables in NOVA to use this input format.

MAXRAW	The number of (floating point) values to be read from the file for each event.,
RAWFP(MAXRAW)	A Floating Point data vector into which the data is read.
RAWMIN(MAXRAW)	
RAWSCALE(MAXRAW)	

The input driver will construct the (INTEGER) $RAW vector according to:

$RAW(I) = (RAWFP(I) - RAWMIN(I)) * RAWSCALE(I)

HERMES The HERMES data format. Details of this interface are given in a separate document.

MIDAS The MIDAS data acquisition system used at TRIUMF for the Charge Symmetry Breaking experiment 121 (and a few others). (Just to confuse things, the network-based package from PSI - also called MIDAS - is NOT this format. It is also supported, but is invoked with the command IMODE NET).

NET The network based package from PSI. This mode allows the data being analyzed to be collected on a different machine, and shipped to NOVA by a server process funning on either the local (NOVA) machine or the remote machine. Note that the PSI packagbe is also called MIDAS - it is referred to as NET within NOVA (or sometimes YBOS).

NOVA/VDACS Data generated by the NOVA program itself (using the OUTPUT command in the OPSEQ). NOVA output tapes are identical in format to VDACS tapes, so either IMODE NOVA or IMODE VDACS may be used.

OFDAQ Data in DAQ format, but read from a tape/disk file rather than an online process.

ONLINE This format allows analysis of events generated by an online data acquisition process running in a Starburst J11. The data format is in fact identical with that for NOVA and VDACS — however, this mode also specifies to NOVA that the data is coming from a real time data acquisition process (VDACS) rather than from a tape/disk file. Note that you use IMODE CODA, though, to specify an "online" source in the CODA system.

IMODE ONLINE is not supported when running under Unix.

SAL This format accepts data generated at the Saskatchewan Accelerator Laboratory.

TEST This generates pretend data. The event type is always 1, the length is always 100, and $RAW(I) = I (i.e. the value of each raw data word is equal to its index in the raw data vector).

USER User-defined event format. The user is expected to provide subroutines (and link them using the NOVALINK command) to manage extracting events from some input source and presenting them to the analysis package in NOVA. Details of the routines which must be provided are given below later in this document.

VDACS This format allows analysis of data tapes (or disk files) written by the TRIUMF online acquisition system. This command is obsolete — you should use IMODE NOVA instead.

YBOS In this format, event data is assumed to be contained in a series of “banks”. Each bank is preceded by a (20 byte) header containg the bank name (4 characters exactly), a type, length and various other system information. These headers appear in the $RAW data vector, and your analysis code must be prepared for them. There is currently very little additional support for YBOS data banks in NOVA - typically all that is required of the analysis code is that it be aware of these bank headers and ignore them.

The NET package from PSI (described above) typically produces YBOS format by default.

It is a relatively simple job to include other data formats into NOVA (although probably not one that I would like to allow the unwashed user to tackle). If you would like NOVA to be able to analyse data in a different format, let me know and I will write an input driver for your particular input format. A brief description of the routines required (for those of you who really want to tackle it yourself) is given below.

- *INTEGER*2 name {(size)}, {name ...}*

Declares one or more variables to be of type INTEGER*2. Variables may be dimensioned by this command also (see the DIMENSION command above). The "*2" is required (since the default for Integer variables is INTEGER*4), but the keyword INTEGER may be abbreviated (INT*2, I*2, etc. are all recognized). This is useful primarily for large data vectors, to save space. Single, undimensioned variables are in fact always stored as INTEGER*4 variables regardless of how they were declared, so there is no saving in storage space. In some instances, it is convenient to declare a simple variable as INTEGER*2, however (e.g. if you want to take advantage of the fact that 32767 + 2 evaluates to -32767 because of "wrap-around"). Such "tricks" are not encouraged, as they easily lead to incomprehensible code, but sometimes there is no reasonable alternative.

Examples:

INTEGER*2 X, Y, Z
INT*2 I2VECTOR (100)
I*2 I2VALUE

- *INTEGER*4 name {(size)} {name ...}*

Declares one or more variables to be of type INTEGER*4. Variables may be dimensioned by this command also (see the DIMENSION command above). The '*4' part is optional, since Integer variables are INTEGER*4 by default.

Examples:

INTEGER*4 I4VECTOR (20)
INT I4VALUE

- *LIST name {name ...}*

Lists the current definition of one or more items. Wild card characters in *name* are supported. Information listed depends on the type of the entry and on the complexity of listing requested. If no *name* is entered, *all* items will be listed (i.e. the default is **LIST ***). The LIST command can be interrupted by CTRL-C to terminate excessively long output.

If *name* contains wild cards, the items will be listed in *alphabetical order*, not in the order in which they were defined.

Qualifiers:

/1D	List only items of type 1D SPECTRUM.
/2D	List only items of type 2D SPECTRUM.
/ALIAS	Lists only those items which are of type ALIAS (i.e User Defined Commands).
/ALL	Normally the listing does not include system-defined variables (**$ALLCONDS, $RAW,** etc.). The **/ALL** qualifier forces these items to be included.
/BRIEF	Only the name is listed (5 per line).
/CMASK	List the conditions which are currently part of the system spectrum condition mask. This will include all conditions mentioned in **/CONDITION=(list)** for all spectra. Currently the maximum number allowed for all spectra is 256.
/CONDITION	List only items of type CONDITION.
/DECIMAL	List all integer values in decimal. This is the default.
/FULL	A complete listing (usually 2 or 3 lines) of everything known about the item is produced. This is the default if *name* contains no wild cards. For vectors (i.e. things which are dimensioned) the /FULL qualifier also lists the values of the entire vector.

If you LIST/FULL $RAW (the raw data vector), the command will complain if the declared size of $RAW is smaller than the amount of data actually present (specified by the current value of $EVENTLENGTH), warning you that data is being lost off the end of $RAW.

/GROUP List only items which are groups. This can be combined with other qualifiers to select only certain types of groups (e.g. LIST/GROUP/CONDITION list only condition groups). The members of the group are NOT listed by this command — use the /MEMBERS qualifier to see the members of a group[g].

/HEXADECIMAL Lists all Integer values in hexadecimal.

/I2 List only items of type INTEGER*2.

/I4 List only items of type INTEGER*4.

/INTEGER List only items of type INTEGER. This is equivalent to the combination /I2 /I4.

/LABEL List only items of type OPSEQ LABEL.

/MEMBERS List the <u>members</u> of groups. This qualifier implies the /GROUP qualifier also (i.e. LIST/MEMBERS is equivalent to LIST/GROUP/MEMBERS).

/NORMAL A one line description of the most commonly needed information is produced. This is the default when *name* contains at least one wild card character.

/OCTAL List all Integer values in Octal.

/ORPHAN Lists only items which are *orphans*. See the DELETE command for a discussion of "orphans".

/PARAMETER List only items of type PARAMETER (either scalar constants defined implicitly - X=143.2 for example), or names which have been explicitly declared using the PARAMETER command.

/REAL List only items of type REAL*4.

/REF Lists *references to* each item rather than the definition of the item. The default here is */BRIEF* (i.e. list only the names of things which refer to a given item). This is useful when the system won't let you delete something, to find out who is still referring to it, or to discover what might be affected if you change the definition of a variable.

/SPECTRUM List only items of type SPECTRUM. This is equivalent to LIST /1D /2D.

/UNDEFINED Lists only those items which are undefined. When you are all done defining things, there must be no undefined entries left or the system will not let you begin analysis (or at least will ask if you are sure that you know what you are doing). This command lets you find out which items it is complaining about.

/VARIABLE Lists only items of type VARIABLE. This is equivalent to LIST /I2 /I4 /REAL (except that PARAMETERS are not included).

Many of these qualifiers may be combined. for example:

LIST/UNDEFINED/1D Lists only 1D spectra that are undefined.

[g] This behaviour is different from earlier versions, where LIST/GROUP listed the members of groups.

LIST/REAL/I4 Lists only REAL and INTEGER*4 variables.

/BRIEF, /NORMAL and /FULL may be used with any other qualifiers.

In previous versions of the program, it was necessary to explicitly DA before using the LIST command, as analysis by the subprocess continued in parallel and you would therefore get things in the list which belonged to different events. This is now no longer a problem — the LIST command does its own implicit DA before it starts the list and EA after it is done (i.e. analysis is "temporarily" suspended during the execution of the LIST command).

Examples:

LIST/VAR X*	List all variables starting with X
LIST	List everything except system variables
LIST/BR/1D	List names only of all 1D spectra

- *LIST/IF n1 n2*

List only the IF statements in the OPSEQ between lines n1 and n2 (inclusive). If n1 and n2 are omitted, all IF statements in the OPSEQ will be listed. This listing differs from the LIST/OPSEQ (below) in that only the IF statements in the OPSEQ are listed. In addition, the software scalers for each IF statement are also listed (how often the statement was executed and how often the *expression* was TRUE) unless the /NORMAL qualifier is also used.

Qualifiers:

/NORMAL	List only the IF statements but not the software scalers. The default is to list the values of the software scalers as well.

- *LIST/OPSEQ n1 n2*

List lines n1 through n2 (inclusive) of the OPSEQ. If n2 is omitted, only line n1 will be listed. If both n1 and n2 are omitted, the entire OPSEQ will be listed.

The format produced bu this command has changed. Each line now contains also an “IF Level” indicator, giving the number of IF statement blocks within which it is contained.

Qualifiers:

/FULL	With each IF statement, list the values of the software scalers for the IF statement. The default is to list the statements only.

- *LOAD filename {name, name, ...}*

This command loads tables and / or spectra from a file created with the DUMP command. If no names appear in the command, everything in the file will be LOADed.

Dump files may be either in BINARY or ASCII (see the DUMP command). If the extension on *filename* is .NBI (Nova BInary file), LOAD will attempt to treat the file as a Binary dump. If it is .NCM (Nova CoMmand file), it will attempt to treat it as an ASCII dump file. If the extension is neither of these (or there is no extension) it will first attempt a binary LOAD — if this fails it will then attempt an ASCII load.

For an ASCII file, you can either *LOAD* the file, or use *@filename* to execute the commands in it as if they were entered from the keyboard. The difference is that *@filename* will add the definitions to any existing ones, while *LOAD filename* will delete everything that is there first (i.e. it will overwrite the existing tables). Note that if you use *@filename* and the file contains the OSPEQ (as it usually will), you will end up with 2 copies of the OPSEQ since what is in the file is simply added to what you already have in defined in memory. This is not usually what you want, although there might bd instances where this is useful.

If names appear on the command line, only definitions which match those names will be loaded (wild cards are allowed), implementing the long-promised "selective LOAD" capability. For example, the command LOAD file X* will load only those definitions for things which begin with X. This selective load capability is currently only supported for BINary dump files, however.

Earlier versions of (Binary) DUMP files are automatically converted to the most recent version when they are LOADed (you will see a descriptive message when the LOAD command tries to perform this conversion). If problems occur, please save the offending files so that I can fix the problem.

Qualifiers:

/ADD This is supported for BINary files only, and adds the contents of the file to what is currently in memory (allowing you to sum results from individual runs). Counts in all spectra and software scalers for all conditions (both those explicitly declared and those associated with IF statements and spectra) are summed. Variables are NOT added together, so that (for example) scaler TOTALS are not correct[h].

/ALL Load both table definitions and spectra (if present) from the dump file. This is the default.

/SPECTRA Load only the spectrum data but not the table definitions. It is the user's responsibility to ensure that the spectrum data corresponds to the table structure that he currently has. Use of this qualifier is not recommended.

/TABLES Load only the table definitions but not the spectrum contents.

- *LOCK*

By default, any process is allowed to change NOVA definitions / tables if it is allowed to MAPTO it[i]. If you wish, you can prevent other processes from doing this, and become in effect the exclusive owner of the subprocess (a word of caution — the reason that this change was made was for convenience in an online environment. You can change this default using the LOCK command if you want, but be prepared for a somewhat less convenient mode of operation).

The STATUS command indicates who currently has exclusive write access to a particular subprocess (or displays the string AVAILABLE if no-one does).

If you attempt to alter the tables of a process which has been locked by someone else, NOVA will issue the error message *No write access.*

A locked process is released (and made available to everyone else) with the UNLOCK command. A different process can then issue a LOCK command to prevent write access by others.

In some circumstances, it convenient to be able to force your way into obtaining write access to a NOVA subprocess. For example, you might find yourself at a remote location (e.g. on the MRS platform) and need to get access to a running NOVA which "belongs" to a terminal somewhere else. In such circumstances, the BREAKIN command is available to allow this.

- *MAPTO {name}*

It is possible to have several NOVA subprocesses running at once, limited only by the constraints imposed by the operating system (e.g. subprocess quota). Only one of them is "current" at a given time, however (that is, they can all be busy

[h] I am working on a solution to this problem.

[i] This behaviour is different than in earlier versions NOVA.

analysing data, but the user sitting at a terminal can only communicate with one of them at a time). This command allows you to change which NOVA subprocess you are talking to.

If *name* is given, the system will MAPTO the subprocess named *NOVA_name*. If *name* is not entered, the system will list all of the NOVA subprocesses which belong to you (the one which is now current will be shown flashing), and you may then select which one you want (by number).

When you begin a new subprocess (with the VMS NOVASTART command), it becomes the "current" subprocess by default, so it is not necessary to explicitly MAPTO it.

When you MAPTO a subprocess, it remains the active one until you explicitly change it (*even if you exit NOVA to DCL and then re-enter NOVA, so be careful*).

While talking to VMS, the NOVAMAP command performs the same function.

- *MOUNT*

The MOUNT command logically mounts a magnetic tape. Note that this is not the same as the VMS MOUNT function. If a tape has been MOUNTed under VMS, you still should issue the NOVA MOUNT command before you use the tape (in fact, the functions carried out by the MOUNT command are automatically carried out the first time you access the tape if you have not explicitly issued the MOUNT command, but it is a good idea to do it explicitly anyway).

The MOUNT command rewinds the tape and reads the first record on it. If it is of the right format (exactly 80 bytes long, with the first 4 bytes 'VOL1') then NOVA assumes that the tape is labelled, and it sets itself up to be able to access the data on tape by file name. If the tape is unlabelled, then all access to data on the tape must be by file number (using the SKPF command) which is much less convenient.

Once a tape has been MOUNTed (either explicitly with the MOUNT command or implicitly), then NOVA assumes that all input files which are to be analyzed reside on the tape (that is, the input *device* is assumed to be the tape drive). If this is not what you want, you must explicitly DISMOUNT the tape before you can access input files as disk files.

For example, to analyze a run EXPT00001RUN12345 on tape, you would issue the following NOVA commands:

MOUNT MTAPE
OPEN EXPT00001RUN12345

Without the MOUNT MTAPE command, OPEN would assume that EXPT00001RUN12345 was a disk file, and would search for it there.

Note that the OPEN command only searches forwards on a tape for a given data file. If the tape is currently positioned after the run that you want (it is up to you to know what is on the tape), you have to REWIND or SKPF backwards first.

- *MTABORT*

If you have made a mistake in mag tape handling (e.g. you forgot to REWIND, and the OPEN command is busy searching for a file that it isn't going to find), you can issue the MTABORT command to terminate the previous command. Be patient! Tape handling is actually carried out by the NOVA subprocess (it "owns" the tape), and as it carries out tape functions, it periodically checks to see if the command has been aborted. So a response to the MTABORT command may take a few minutes.

- *NEW*

This command erases *everything* in the tables. The system will ask you for confirmation unless the /OK qualifier is entered.

Qualifiers:

/OK Do not require keyboard confirmation of the deletion (i.e. just go ahead and do it).

- *NOECHO*

This command turns off the automatic echoing of commands in @-files (see the *ECHO* command). You can also use the SET NOECHO command perform the same function.

- *OPEN {/INPUT} {file/device}*

This command opens a file / device in preparation for analysis. The /INPUT qualifier is optional (i.e. the default is /INPUT if no qualifier is present). Depending on the input mode / format (set with the IMODE command), *filename* may or may not be required (or even allowed).

The input mode (IMODE) must be set before you enter the *OPEN* command. If a file was open previously, it must first be *CLOSEd* before another one can be opened.

If the input mode is *ONLINE* (i.e. talking to VDACS), the system will attempt to open the default channel GRA1: unless the user requests a different channel in the OPEN command.

If the input mode is *TEST*, no filename is permitted.

If the input mode is *CODA* and *filename* is not given, the system will attempt to read events from a shared memory region on the same machine as you are running NOVA. If *filename* ends with a '**:**' character, the system will read events from a shared memory region on the computer named 'filename' via RPC (Remote Procedure Call)[j].

If the input mode is YBOS, the “filename” is interpreted as follows:

A “normal” filename is interpreted as a disk file. The program attempts to open this file and read events from it.

The filename “local” (case insensitive) attempts to read events from an oline MIDAS server running on the local machine (i.e. no network connection is involved).

A filename of the form “hostname:experiment” (i.e. something with a colon in it) willcause NOVA to attempt to connect to a MIDAS server on a remote machine “hostname”. If “experiment” is blank, the environment variable EXPT_NAME is used instead. If EXPTNAME is not defined, the remote MIDAS server will consult its file ‘exptab’, which should contain only a single entry (NOVA does not allow you to interactively select an experiment from exptab).

In all other cases (where you are presumably analysing old data) the *file* parameter specifies the device (i.e. mag tape) or disk file to open.

A minor problem occurs running under Unix. Unix filenames (other than simple files in your current default directory) contain the '/' character (as in '/usr/users/pewg/datafile.dat'). The NOVA command parser (at a very early stage in parsing a command) treats things which start with '/' as command qualifiers, and hence does not understand something like this. So, to enter a file name which contains a '/', you can (from outside NOVA) enter a command like:

```
setenv DATAFILE /usr/users/pewg/datafile.dat
```

You can then, from within NOVA, say

[j] This has never in fact been tested.

OPEN/INPUT DATAFILE

and everything will work as you expect[k]. In particular, when analyzing directly from mag tape, you should

```
setenv MTAPE /dev/nrmt0h
```

As an alternative, you can set up a link in your default directory to the file that you actually want. All references to this 'link' file are treated by Unix as identical to the original file. For example,

```
ln -sf /usr/users/pewg/datafile.dat DATAFILE
```

has the same effect as the 'setenv' command above — all references to DATAFILE actually refer to /usr/users/pewg/datafile.dat. An advantage of this approach is that you can do this from within the NOVA DCL command, and so never have to get out of NOVA and then back into it again. Conversely, environment variables must be set before you enter NOVA and cannot be changed from within a given NOVA session.

As a final way around the problem, you can explicitly enter the /INPUT qualifier. The OPEN command is clever enough to realize that evrything after the first qualifier is a file name, and all special characters in it will be ignored. So you could also enter

OPEN/INPUT /usr/users/pewg/datafile.dat

- *OPEN/OUTPUT filename*

This command opens a file / device to write events produced by the OPSEQ OUTPUT command. The /OUTPUT qualifier is required (the default for OPEN is /INPUT). Filename is either a tape unit or a disk file.

Opening an output file effectively "enables" output from the OPSEQ (i.e. if you don't want to produce output any more, the easiest way to do it is to CLOSE the output channel — it is not necessary to remove all of the OUTPUT statements from the OPSEQ).

Note that this output device is not the same as the one that the online VDACS is using to log raw event data. The NOVA output device contains only analyzed events, and is intended to be used for event skimming (for example), not data logging.

See the comments in the OPEN/INPUT section above for ways of dealing with '/' characters in Unix filenames.

- *PARAMETER name {name ...}*

This defines one or more names to be PARAMETERS (i.e. things which NOVA doesn't try to calculate before it uses them). This statement is most often used for dimensioned variables, since simple (scalar) parameters are implicitly defined as such by simple entering

name = constant_value

A vector *must* be declared as a PARAMETER before you can set its values with simple assignment statements (as in **X(n) = constant_value**).

These are not really PARAMETERS in the FORTRAN sense of the word — Constants would be a better term (in fact, in earlier versions of NOVA they were called CONSTANTS). I changed the term to PARAMETER primarily because

[k] Essentially what happens is that NOVA treats the filename argument of the OPEN command as a logical name (environment variable in Unix) and attempts to translate this name vefore it tries to open the file.

Constant is too close to Condition (in older versions of NOVA, you couldn't say LIST/CON because CON could mean either CONSTANT or CONDITION).

- *PAUSE*

This command stops analysis by the subprocess. Analysis will stop at the end of the current event, and will remain stopped until another *ANALYSE* or *EA* command is entered. *DA* (Disable Analysis) may be used as a synonym for PAUSE.

In earlier versions of the system, it was necessary to PAUSE before altering definitions in the NOVA tables (in fact, NOVA did it for you automatically whether you liked it or not). This is no longer true (in fact, NOVA still does PAUSE while you are actually defining something (so that the subprocess doesn't get confused as you are altering table structure), but it starts up again automatically when the command has completed).

- *PROFILE {ON/OFF}*

NOVA contains a primitive (but still useful) PROFILE facility which allows you to discover where your NOVAMAIN.EXE program is spending most of its time (you might use this to decide which pieces of your code need to be optimized). The profiling interval is 10 msec.

If profiling is turned on, the program generates a disk file named 'movamain.mon' in which the profiling information is stored. When profiling is disabled (it adds a non-negligible overhead, so you should turn it off when in "production" mode), this file will contain a list of bit-patterns corresponding to various parts of the code and a count of how often the profiler found the program in this place (a rough indication of what fraction of the time the program spends here). Much of this is useful only to me (time spent histogramming, for example). The parts useful to Users are those entries with the bit pattern '4000'+n, which indicate that the program was in User Function USRn.

This disk file is created / updated by the PROFILE OFF command (i.e. it is not available - or at least not current - while profiling is enabled).

Qualifiers:

/OFF	Turn profiling off.
/ON	Turn profiling on.

- *QUIT*

QUIT is a synonym for END, EOF or EXIT. It is used primarily to terminate OPSEQ insert mode. If entered in COMMAND mode (i.e. while you are talking to the NOVA Command Line Interpreter), you will be returned to VMS DCL. Note that it is not terribly drastic if you do this by mistake. The analysis subprocess continues analyzing, and all changes that you have made to the NOVA tables are saved. You simply have to enter the NOVA command again and you are right back where you were.

Even though (so far at least) Q is the shortest legal abbreviation for the QUIT command (i.e. no other commands start with Q), this single letter short-form has been explicitly disabled. You have to enter (at a minimum) QU to exit NOVA with this command.

- *READ vector filename*

The READ command reads the values of a vector from a disk file. *filename* is assumed to specify a series of values which are assigned to consecutive elements of *vector* (which must be dimensioned and declared to NOVA as a PARAMETER). Values are read from the file in free-format — with blanks, tabs, commas or carriage returns separating them. The only "restriction" on the format is that any line which begins with '!' (in column 1) is treated as a comment line and is totally ignored. Reading continues until either the end of file is reached or all elements in *vector* have been assigned values.

For example, if a vector JUNK had been declared as INTEGER*4 JUNK(5), the following file might be used to set its values

```
!       Comment — remember that the first element is JUNK(0)
!       so this file should contain 5 numbers to set everything
!       from JUNK(0) through JUNK(4)
!
14,      16          12
5
! and finally
17
```

● *REAL{*4} name {(size)} {name ...}*

This declares one or more variables to be of type REAL*4. Variables may be dimensioned by this command also (see the DIMENSION command above). The '*4' part is optional (since only REAL*4 variables are implemented).

Examples:

REAL*4 XVECT (20)
REAL X,Y

● *REWIND*

This command simply rewinds the tape unit. It should be used only if the input device really is a tape drive — set with the MOUNT command — it <u>will not</u> rewind a disk file. To position a disk file to the beginning, you should close and re-open it.

● *SET*

Sets various parameters defining the operating environment for the system. *Note that most parameters set with this command are not saved from one session to another* (there are two exceptions, BYTESWAP and DECWGEOM), and are <u>not</u> stored in the NOVA tables (that is, they are "private" to your process rather than shared by everyone who does a MAPTO). Hence they must be re-entered every time you type NOVA (or perhaps entered in your NOVA_STARTUP file, where they will be executed automatically every time you start a NOVA session). They are intended to set the operating environment for a particular user / process without changing it for other users (i.e. they are essentially local variables for each process).

Qualifiers:

/BYTESWAP=n Defines how data read from an input tape / file is to be manipulated before being passed to the analyzer. Options are:

0	No change. This is appropriate if you are analyzing the data on the same architecture of machine as it was collected. This is the default.
1	Bytes within each INTEGER*2 word are swapped. This is appropriate if the machine architecture is different (big/little endian) and the input data is treated as INTEGER*2 words.
2	The two INTEGER*2 words within an INTEGER*4 longword are interchanged, but the bytes within each INTEGER*2 word are left intact.
3	Both words and bytes are interchanged.

/CONFIRM=option Defines under what conditions NOVA will request confirmation before changing a variable definition. The options are:

ALL Equivalent to both CHANGE and NEW.

CHANGE Confirmation is requested if you attempt to *change* the definition of an existing variable.
NEW Confirmation is required if you attempt to define a *new* variable / spectrum.
NONE (Default). No confirmation is required.

/CRET=option At many points during execution, NOVA may request single character input from the user (e.g. OK TO CONTINUE (Y/N) ?). Normally, the user is expected to respond with a single character followed by a carriage return. If you prefer (and I do, which is why I put this option in), you can set this flag so that the carriage return is not required — just typing the single character is sufficient. The options are:
YES Carriage return is required (this is the default).
NO Carriage return is not required.

/DECWGEOM This option doesn't really belong with the SET command — rather that setting an option for the current session it actually SAVEs it for the next session.

When you open a graphics window on a DECWINDOWS terminal, the window is placed in a default position and given a default size. You can move / resize the window using the mouse. When you are happy with the new look of the graphics window, you issue this command and the system "remembers" this information, and will use it (rather than the defaults) the next time you enter NOVA and open a graphics window.

The information is saved in a VMS logical name (Unix environment variable) and so is not saved from one login to the next. If you wish, you can enter the appropriate information in your LOGIN.COM (or .login) to set this information automatically for all sessions. The format is:

```
DEFINE DECWGEOM "string"     (VMS)
setenv DECWGEOM "string"     (Unix)
```

where 'string' looks something like

```
width x height xpos ypos
```

'width' and 'height' specify the width and height of the graphics window in pixels, 'xpos' and 'ypos' specify the screen position of the upper left corner. Note that 'string' must be enclosed in double quotes as it contains blanks. If you want to do this, it is probably easiest to first set the option in NOVA, then exit NOVA and look at the definition (SHOW LOGICAL DECWGEOM or printenv DECWGEOM). Then enter the definition in the appropriate login file.

/DISPLAY=option Defines the type of display terminal you are using (only for an XWINDOWS terminal - in particular these don’t work for a DECWINDOWS terminal type).
COLOUR (2,3) 2D displays will use colours (as well as size of squares) to indicate the number of counts in a channel.
CREVERSE (6,7) A combination of COLOUR ane REVERSE.
MONOCHROME (0,1) No colour information will be included in 2D plots.
REVERSE (4,5) Like MONO except foreground / background colours are reversed (white lines on a dark background).
n (0 - 7) There is an inconsistency in the treatment of X-windows displays between PC’s (those running Linux at least) and (NCD) X-terminals. The problem most often manifests itself when diplaying the cross-hairs - settings for an NCD terminal result in the cross-hairs being invisible on a PC and vice-versa. To overcome this problem, you can (for each display type listed above) specify it also as one of two integers. For example, if you want a MONO display, but then notice that the cross-hairs are invisible, you can try SET /DISP=0 and (if that doesn’t work) SET /DISP=1. One of them should work properly (if neither does, let me know).

Colour displays are implemented only for 2D plots. In addition to making the size of each box proportional to the counts in a channel, different counts are also drawn in one of 8 different colours. This has proved convenient for some of the TRINAT data, but is of limited use in general (at least I think so).

/ERROR=option Defines the format of error messages produced by NOVA.

SHORT (Default) produces a cryptic (usually 1 or 2 words) error message. This is usually good enough if you are familiar enough with NOVA to know what sorts of things you are likely to do wrong, but it is not terribly descriptive.

LONG produces a longer (1 or 2 lines, usually) error message. SET /ERROR=LONG also will echo the most recent error message in its long format, so that if the short one really is too cryptic, you can use this to find out what it really means. Make sure that you do this before you type another NOVA command as the next command will overwrite the previous error code — only the most recent error is remembered.

/HPBMARGIN=n Normally information plotted on the printer is scaled to fill a complete (8½ x 11) sheet of paper. This qualifier (and HPLMARGIN described below) allow you to override this default and leave a margin at the bottom and/or left of the paper (e.g. to leave room for a notes at the bottom of the page). ‘N’ is an integer (in tenth’s of inches).

/HPCOMPRESSION=n Some (not all) HP printers support some form of data compression, whhich can significantly reduce the amunt of data which must be sent to the printer. N=0 (the default) sends normal uncompressed data. N=1 and n=2 specify different levels of compression (n=2 compresses more - be warned though that I have never made this work properly - I don’t know if is a function of my particular printer, a bug in the program or a misunderstanding on y part of how it is cupposed to work).

/HPLMARGIN=n See the discussion above for /HPBMARGIN.

/HPPAGE=n This defines the number of (logical) pages of plots to put on one physical piece of paper. It is valid only if the printer type (SET /PRINTER) has been set to one of HPPCL, HP75M HP100, HP150, HP300 or PSCRIPT. Legal values of ‘n’ are 1,2,3,4,6,9.

/INPUT=option Defines how input lines are read from your terminal.

LOCAL (Default). Input is read 1 character at a time. This mode is <u>required</u> for such things as command recall / editing using the cursor keys. It is also the only mode which supports the EDIT command for variable definitions.

REMOTE Reads input 1 *line* at a time. Command line editing is still available (if supported by your <u>local</u> machine, which it nearly always is). It is less convenient, since command recall is not available. However, when logging in over a network from a remote site, single character I/O can be *extremely slow* (any of you who have tried to use the EDT editor over the network will know what I mean). You can switch from one mode to the other at will (i.e. you can SET /INPUT=option whenever you want).

/MAYPROCESS (Default). VDACS (IMODE = ONLINE only) will pass events to NOVA for analysis only if there is time (i.e. NOVA will be kept as busy as possible, but acquisition will <u>not</u> be slowed down just because events are not being analysed quickly enough). Note that event types which are marked MUSTPROCESS in your TWOTRAN program will all be passed to NOVA and analyzed regardless of this option.

/MUSTPROCESS VDACS (IMODE = ONLINE only) will ensure that <u>all</u> events generated by the acquisition program will be passed to NOVA for analysis (<u>slowing down acquisition if necessary</u>). Note that this could be *dangerous*, since if NOVA cannot keep up, acquisition will be slowed down (and VDACS sometimes gets annoyed if it runs out of data buffers).

/PRINTER=type Defines the type of printer you are connected to (so the *HARDCOPY* command knows how to generate a hardcopy of the screen). Current options are:

CHPPCL	A combination of COLOUR and HPPCL (described below). Note though that Colour printing is not (yet) supported.
CPSCRIPT	A combination of COLOUR and PSCRIPT (described below). Colour printing is not (yet) supported.
EPSON	Locally connected to an Atari 1040.
HP75	HP bitmap at 75 dpi resolution.
HP100	HP bitmap at 100 dpi resolution.
HP150	HP bitmap at 150 dpi resolution.
HP300	HP bitmap at 300 dpi resolution.
HPPCL	HP printer using PCL (Printer Control Language). The resolution will be the default resolution of the printer (usually 300 dpi). Depending on the complexity of the plot, the amount of data sent to the printer may be substantially smaller (and hence the plots will appear more quickly).
HPLASER	Default — Locally connected to an Atari 1040.
PSCRIPT	PostScript.
REMOTE	Plots will be generated on the VAX or Unix printer queue HPLASER.

Some discussion is needed to clarify the above printer types. HPLASER refers to an HP LasetJet connected directly to the parallel port of an Atari 1040. HPPCL, HP75, HP100, HP150 and HP300 refer to a remote printer (like REMOTE) accessed by the printer queue name HPLASER. The difference is in the format of the data sent to this printer. REMOTE should be used if plots are generated with GPLOT (see SET TERMINAL). The vaious HP types are used if the plots are generated using the XWINDOWS (not DECWINDOWS) display driver. If an HPPCL printer is specified, plot information is transmitted using a special HP language called HPPCL (which uses, for example, a special ESCape sequence to draw a line between two points). A potential difficulty with using PCL is that if the plot becomes very complicated (lots of lines), the printer can run out of memory and be unalbe to process the PCL commands (typically you get an Error 20 on the front panel). On the other hand if the plot is not too complicated, the amount of information which must be transmitted to the printer can be significantly smaller (and hence plots appear more quickly). In addition, PCL always uses the best resolution available for the printer.

HP75 etc. generate (internally - within NOVA) a bitmap at the specified resolution and send this to the printer. Very little processing needs to be done (by the printer) in this case, so the Error 20 problem is less likely to occur (if it does, try the next lower resolution). On the other hand, a complete bitmap of an entire printed page must be sent to the printer (> 1 Mbyte at 300 dpi), so the time needed toactually transfer the data to the printer can be long.

As well, I have included the option of specifying a colour printer (CHPPCL for PCL and CPSCRIPT for PostScript). In fact, colour printng has not yet been implemented (it is anyway used only for 2D plots - see SET /DISPLAY). On black-and-white printers I fake it by using different levels of gray shading for different colours. The results (in my opinion anyway) are far from impressive, but theoption is there if you want to use it.

/PROCESS=option This is an alternative format for specifying MAYPROCESS or MUSTPROCESS. The options are:

MAY	(default)
MUST	

/SNAP=option This controls the action of the CURSOR command when you select a channel (either a Left or Right channel, or just a channel whose counts/position you want to look at). The options are:

OFF	The crosshairs will be left where they are.
ON	(default) The crosshairs will "snap" to the centre of the selected channel (thereby showing you the channel that you "really got"). For 2D spectra, the crosshairs will be positioned to the centre of the box defining this channel.

/TERMINAL=type Defines the type of terminal you are sitting at. The options are:

ATARI	
CIT467	C-Itoh 467.
DECWINDOWS	This supports both the console on a workstation and an X-terminal (e.g. NCD) on a network.
PT100G	Plessey.
SEIKO1105	Seiko.
T4010	Textronics 4010
T4107	Textronics 4107.
VT241	DEC VT241.
VT640	(default).
XWINDOWS	X windows terminal (not the same as DECWINDOWS).

This option is currently used only for drawing histogram displays. If the terminal type has been set to **ATARI**, the system will use the ST640 fast histogramming package to draw the plots (this can be over-ridden in the *DSP* command by explicitly specifying /VT640). All other terminal types (except XWINDOWS) specify that the TRIUMF graphics package (GPLOT) should be used to draw the plots. Thus, if your terminal really is an Atari, you could still specify **/TERMINAL=VT640** if you want the plots to be drawn without using the ST640 fast display package). This is <u>much</u> slower that ATARI mode, but is <u>required</u> if you subsequently want to print the screen on a REMOTE printer queue with the HARDCOPY command.

Note that DECWINDOWS and XWINDOWS are <u>NOT</u> the same thing. DECWINDOWS uses the TRIUMF GPLOT package. XWINDOWS uses a special (much faster) driver optimized for an X-windows display (either NCD X-terminal or a workstation). Many of the newer options included in this release of NOVA (COLOUR/REVERSE display, HPPCL/PSCRIPT printers) are supported only for the XWINDOWS display, not DECWINDOWS.

/TIMEOUT=n When the user process sends a request to the subprocess, a response is expected within a relatively short time (typically a few seconds). If no response is received, the user is warned that the subprocess is not responding (perhaps it is "hung", or perhaps it has made a fatal error and been terminated).

Magnetic tape operations on the other hand often take a <u>much</u> longer time to complete (for example, the first read of an Exabyte tape can take 30 seconds or more). For all operations involving magnetic tape, therefore, the user process is prepared to wait a much longer time before declaring that there is a problem. The default wait (60 seconds) can be changed with this command — n is the time interval (in seconds) to wait.

- *SHOW {name}*

This lists the current values of the options defined using the SET command. If *name* is given, only those options which match it (wild cards are supported) will be listed. If *name* is not given, all parameters which can be set with the SET command will be listed.

- *SKPF {n}*

This command is used to position a magnetic tape. If *n* is positive, the tape will be spaced forwards over n file marks. If *n* is negative, it will be spaced backwards over n file marks <u>and then spaced forwards over 1</u>. (If you are spacing backwards over many files, or to somewhere near the beginning of the tape, it may be faster to use the REWIND command and then SKPF forwards). If *n* is not entered, it defaults to n=1 (i.e. the tape is spaced forwards over 1 file mark). Note that the additional space forwards when n is negative means that it all cases, the tape will be left positioned immediately <u>after</u> a file mark, ready to begin reading data.

Tapes can be either labelled or unlabelled (VDACS creates labelled tapes by default, CODA produces <u>only</u> unlabelled tapes). A labelled tape actually contains 3 "files" for each run. The first file consists of a series of short records (header labels) in a standard format which identify the *name* of the file — for VDACS tapes the name is something like EXPTxxxxxRUNyyyyy. The second file contains the data itself. The third file contains more labels (called trailer labels).

NOVA knows whether a tape is labelled or not because when it is first opened (or MOUNTed), it reads the first record on tape and tries to decode it as a standard label. The SKPF command handles labelled tapes properly. That is, if a tape is labelled, the number *n* actually refers to the number of runs to be skipped, not the number of physical file marks. For example, if the tape were positioned at the beginning of tape (just after a REWIND, for example), a SKPF 1 command would actually skip over 4 physical files — the 3 files in the first "run" and the header labels of the second run. In all cases, the SKPF command will leave a labelled tape positioned so that the first thing which will be read is the first piece of "real data".

Qualifiers:

/LOG

Normally the SKPF command does its work silently. If you want to monitor its progress (perhaps to discover where on the tape an error is occurring), you can use the /LOG qualifier. A message will be output to the terminal screen every time a file mark is encountered.

/RAW

In some cases (either because there are errors at the beginning of the tape or the labels are somehow not written in a standard format), NOVA may not properly recognize that a tape is labelled (it tells you when the tape is MOUNTed whether it thinks it is labelled or not). In such cases, you will have to take care of the label records yourself. You just need to remember that the "real data" resides in files 2, 5, 8, ... on tape — files 1, 4, 7 ... contain the header labels and files 3, 6, 9, ... contain the trailer labels.

Additionally, there may be instances where there are errors in the middle of a tape which prevent correct label processing. You can force the SKPF command to treat a tape as unlabelled by using the /RAW qualifier. Again, it is then your responsibility to keep track of which files on tape contain label records and which ones contain real data.

- *SRUN {...}*

The SRUN command is included to allow the NOVAMAIN subprocess to perform some (unspecified) function at the beginning of a run. It causes NOVAMAIN.EXE to call the (user supplied) subroutine USRSRUN, passing to it the remainder of the command line (if any). It is similar to the USREXE command (in fact identical, except that it is intended to be used only at the start of a run).

- *STATUS*

Shows you the status of the analysis subprocess (number of spectra defined, number of events analysed, etc.). This page also shows the current "owner" of the subprocess (i.e. the process which is allowed to change things) or the keyword AVAILABLE if no one currently has LOCKed it. The display updates every 0.5 seconds. Hit any key on the keyboard to terminate this command (note that the character you type is "gobbled up" by the STATUS command).

The normal "system status" requires the top 6 lines of the page. Below this, the user is free to insert up to 51 variables of his own to be displayed on the status page (these might include counts of different event types and/or conditions, for example). Which variables are displayed here is defined by including them in the variable group $STATPAGE. They are put on the page 3 per line from left to right. If the variable is a CONDITION, the value displayed will be the number of times it was TRUE (i.e. the ‘TRUE”’ part of its software scaler). For normal variables, the current value will be displayed.

- *SUBGROUP groupname member {member ...}*

This command removes members from the group *groupname*. It is the inverse of the ADDGROUP command. If an item is part of a group (e.g. a spectrum), it must be removed from the group using the SUBGROUP command before it can be deleted (since the group "refers" to it).

Wild cards are supported in the SUBGROUP command.

- *SWRITE specname filename*

Writes the contents of the spectrum *specname* into the disk file *filename* in ASCII (i.e. human readable form). If *filename* does not contain an extension, the default .SWR will be used.

The files are human readable, but the format is not particularly obvious. The following information is contained in the file. Note that (for 2D spectra), "X" refers to the horizontal axis and "Y" refers to the vertical axis.

(Apologies — it is much easier to explain this if I switch into "computerese" for a bit — talking about "records" and so on).

begin {computerese}

There are four kinds of "records" in the file.

TITLE Contains the type of spectrum (1 = 1D, 2=2D) , the name of the spectrum and the RunNumber.
FORMAT (I2, 1X, A12, 3X, I12)

AXIS Contains the number of channels along the axis, the lower and upper limits, and the name of the variable.
FORMAT (I6, E16.7, E16.7, A12)

STAT Contains "statistics" for a given row (or column for 2D) of the histogram. Sum(X), sum(x**2), minval, maxval (minval is the minimum value you ever tried to histogram, maxval is the maximum value you ever tried to histogram. If all of your counts appear in the underflow or overflow channels, these values can help you decide what your histogram limits should be to see the data).
FORMAT (D16.7, D16.7, E16.7, E16.7)

DATA Contains the data for a given horizontal (X) row of the spectrum. There will be (#X channels + 2) values, written 5 per line (thus, there may be several lines in this "record", and the last might not have all 5 values in it). If the number of X channels is 12 (and hence the number of X values written is 14 including the underflow and overflow channel), there will be 2 lines with 5 values and 1 line with 4 values). The first value is the underflow channel, the next (#X channels) values are the "real data", and the last value is the overflow channel. The number of (5-element) lines in this DATA record can be calculated from information in the AXIS record.
FORMAT (5E16.7)

The files produced by SWRITE look like:

1D **TITLE** record
AXIS record for X axis
STAT record for 1D spectrum
DATA record for 1D spectrum

2D **TITLE** record
AXIS record for X axis
AXIS record for Y axis
STAT records for the vertical (Y) columns of the spectrum. There will be (#X channels + 2) of these records. The first is for the X underflows, then (#X channels) of "real data", and the last is for the X overflows.
STAT records for the horizontal (X) rows of the spectrum. There will be (#Y channels + 2) of these records.

DATA records. There will be (#Y channels + 2) of these. The first is Y underflows, then (#Y channels) or "real data", and the last is for the Y overflows. Remember that each of these "records" might contain several lines of 5 numbers each, as described above for a 1D spectrum.

end {computerese}

Qualifiers:

/PLOTDATA Information written to the file will be in a format which can be understood (and retrieved) by the PLOTDATA program. For a 1D spectrum, you retrieve the data with the PLOTDATA command

READ filename X Z

For a 2D spectrum, the corresponding PLOTDATA command is

READ filename X Y Z

- *TITLE {string}*

If string is given, this command sets the TITLE to be this string (80 characters maximum). If string is absent, the current title will be listed on the terminal.

This title is purely descriptive, and has no other function in the NOVA system.

- *TTOUT {filename}*

The TTOUT command allows you to re-direct output which would normally appear on the screen to a disk file. This might be useful, for example, to capture the output of the LIST command.

If ‘filename’ is not given, terminal output reverts to the screen.

For example, to get the values of all CONDITIONS into a disk file at the end of a run, you could enter the sequence of commands:

```
TTOUT runsummary.dat
LIST/COND
TTOUT
```

- *UNLOCK*

The UNLOCK command releases exclusive access to a NOVA subprocess (acquired with the LOCK command), and makes it available to everyone. Another process may then LOCK it if desired.

- *UPDATE*

This command makes a new backup file (NOVA_xxxxxx.TBL) containing the current state of the NOVA tables.

Every time you run the NOVA program to change definitions in the tables, the system creates a backup file of the state of the tables, so that if you really mess things up, you can CTRL-Y[1] or EXIT/SAVE to abort and then restore the state of the system from the backup file (the backup file is created only when you actually *change* something, so that commands which don't alter the tables execute more quickly - this is why the first command that you enter in NOVA might take several seconds to execute). If you wish, you can use the *UPDATE* command to create a new backup file at any time. This is just

[1] This doesn’t work with Unix.

like doing a *DUMP/TABLES*, except that the file name is automatically set to NOVA_xxxxxx.TBL (where ‘xxxxxx’ is your process ID).

- *USREXE command*

The USREXE command provides a general way for users to pass commands to the NOVA subprocess for execution. In your NOVAUSR.OLB, you must provide a subroutine which looks like:

```
SUBROUTINE USREXE (ITABLE, CLINE, *)
INTEGER*4 ITABLE (0:*)
CHARACTER *80 CLINE
```

The USREXE command simply causes the NOVA subprocess to call this subroutine, passing it the remainder of the command line in the CHARACTER variable CLINE. It is the responsibility of this USREXE subroutine to decode this command line and perform whatever action is requested.

As one example, you could define a special series of actions to be taken at the beginning of each run (as opposed to each time you EA), and include them all in a subroutine called STARTRUN. You would invoke them with a command

```
USREXE SRUN
```

and your USREXE subroutine would contain code like:

```
IF (CLINE (1:4) .EQ. 'SRUN') THEN
        CALL STARTRUN
        ...
```

If you prefer, you can combine this with the user alias facility to define a new “command” called SRUN

```
SRUN := USREXE SRUN
```

The command line can contain more than just the "command" SRUN — its contents are arbitrary (but your USREXE routine has to be prepared to decode the line — it is passed as an uninterpreted character string). This facility is currently used in the MRS/SASP environment to control the programmable trigger electronics via the TRIG/CALC and TRIG/SET functions.

- *WHICH {USRn}*

The WHICH command is a (very) primitive implementation of code / version management.

Every USRn function in NOVA can contain an 80-character string describing what it does and/or what "version" it is. This is done by including in the source code the statements:

INCLUDE 'nusrfuncs.h'
DATA USRnID /'Description of this USRn function'/

(The 'n' in USRnID should be replaced with that for the particular USR function — USR0ID, USR1ID, USRBID etc.).

If the WHICH command includes the parameter USRn, then this string is typed on the terminal. The intent is that when you make a change to a USRn function, you should also change this character string to indicate which version of the code is currently running (e.g. the above DATA statement in a Real USR5 function might read)

DATA USR5ID /'ADC Gain matching - P.W. Green - Version 1.99 Oct 3 1994'/

What you actually put in here is up to you (except that it cannot exceed 80 characters in length). It should probably contain a date and an author as a minimum.

WHICH (with no arguments) prints out the version of NOVA which you are running (both the NOVAMAIN subprocess and the NOVA CLI you are talking to — these should probably be consistent or you can expect troubles). WHICH * prints out the description of all (36) USRn functions.

- *ZALL*

The ZALL command clears all spectrum counts, and zeroes all software scalers for all conditions, spectra and OPSEQ IF statements. In addition, all variables included in the variable group $ZALLVARS will be set to zero (e.g. scaler TOTS and OLDS might be included in this group).

- *ZERO name {, name ...}*

Zeroes one or more spectra or conditions. Zeroing a condition clears the software scalers for the condition. Zeroing a spectrum clears both the software scalers and the counts in the spectrum.

Zeroing a vector which has been declared as a PARAMETER clears all elements of the vector (e.g. ZERO SCALERS — see the discussion of SCALERS processing below.

You cannot clear the software scalers for an OPSEQ IF statement using the ZERO command - you must use ZALL for this.

Wild cards are supported for the ZERO command. The command *ZERO ** would clear all spectra and condition counts, but is much slower than the *ZALL* command (in addition, ZALL also clears the condition counts for the OPSEQ IF statements — ZERO * does not).

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 (usually) 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 $. Labels are used primarily as the targets of GO TO statements, although they may appear anywhere.

Two labels ($BEG_OPSEQ - the first line in the OPSEQ - and $END_OPSEQ) are supplied automatically. $END_OPSEQ is often used in a GOTO statement after a particular section of the OPSEQ has been executed.

- *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 <u>forces</u> the evaluation of a particular variable (or group of variables — 'var' can be either a single variable or a variable group). It is not normally needed, as variables are automatically evaluated by the system <u>as they are needed</u>. 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), or where you want to force the execution of a USR function. Note that variables are only calculated once (at most) for each event. Thus a complex OPSEQ might end (or begin) 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 <u>would not be affected</u> - any that had not been done already would be evaluated (and their software scalers updated).

Variables can also be calculated by calling a user function (which could have all sorts of other side effects, such as incrementing histograms through user subroutine calls). The EVAL statement can thus 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 <u>without</u> the terminating colon. Backward GOTO's (to an earlier line in the OPSEQ) are allowed (but not recommended - it is possible to get yourself into an infinite loop).

- *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 (expression) 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 - <u>any non-zero value is regarded as TRUE</u>.

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 <u>all</u> 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 members of the group, and *all of them must be TRUE for the GO TO to be executed.*

The keyword GOTO (or GO TO - you may use either one word or two) 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 in your OPSEQ called 'THEN').

There is currently no restriction 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

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 then 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 ELSEIF 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 in this listing 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 NOVA keyword, it is assumed to be the name of a spectrum to be incremented. Note that this implies that OPSEQ 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 *floating point* 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 events.

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 NOVA/VDACS format (i.e. the file can be read by NOVA using the IMODE NOVA 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). It is assumed also that if you have several OUTPUT statements at various points in the OPSEQ that you include sufficient information to allow you to reconstruct later what data belongs with what.

The OUTPUT statement is ignored if you have not opened an output stream with the OPEN/OUTPUT command.

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 environments (which may not support a decent screen editor), you can also "edit" the OPSEQ by line number.

Full Screen (VMS-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 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 editor command file to customize 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).

Full Screen (Unix) Editor

A similar editing facility is also available for Unix. The EDIT/OPSEQ command will invoke your favourite editor as specified by the EDITOR environment variable, and it will be given the filename OPSEQ (Upper case) as its input argument (before this happens, NOVA will copy the current OPSEQ to the disk file OPSEQ). The behaviour of this editor is irrelevant (to NOVA) except that it is assumed that the creation date of the file (OPSEQ) will be updated if you want to retain the changes that you made (as if you typed EXIT in EDT), and will not be changed if you don't (as if you typed QUIT in EDT). NOVA uses the creation date of the file as the only clue as to whether you want to update the OPSEQ or not.

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[m] or the EOF, EXIT or 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)

m Don't use CTRL-Z with Unix.

$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 asks for confirmation first.

See the preceding section on NOVA commands 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 alternatives and wild horses couldn't make you change). Therefore, NOVA tries to let you 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 the more common pitfalls and how you can 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 description of the FORTRAN interface is provided below). In this scenario, the *definition* part of NOVA consists of defining a single variable:

X = USR0 (X)

and the *OPSEQ* contains the single line

EVAL X

This is equivalent to the single 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 it 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, but you do lose something (a lot, I think) with this approach.

2. DACS

In DACS, 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").

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

```
1       (Insert things into OPSEQ)
        IF (.NOT. EVTYPE1) GOTO EV2
        EVAL SCALERS
        GOTO $END_OPSEQ
EV2:    IF (.NOT. 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 start 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 return vector USER_FORTRAN.

```
        USER_FORTRAN = USR1 (USER_FORTRAN, $RAW)
```

Spectra are then defined with a *condition list* on each one, and are incremented by the statement INCR $ALLSPECTRA only if the conditions in the list for a spectrum are all TRUE (presumably, the conditions test some of the values returned in USER_FORTRAN).

```
        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 <u>no</u> limit on the number of conditions in a group - the "maximum 256" conditions applies only to the total number of conditions in the spectrum *condition mask*). A condition becomes a Fatal condition by just adding it to the group FATAL

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

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
ADDGR /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 so), and you lose this flexibility when lots of your code is buried deep 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.

<u>FORTRAN Interface</u>

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

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

X = USR0 (X, $TABLE, $RAW)

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

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

(note that the index of both the NOVA table and the raw data vector starts at 0, not 1). Let's also assume (for simplicity) that this routine just stores things which it calculates 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 to know is the place in NOVATABLE where its *value* is stored (there are lots of other things stored in the tables as well, like its name, whether it has been calculated already for this event etc.). You get its position from the function

ICHPTR = NVVALPTR (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 — its calculation is "hidden" from NOVA inside your USR function. It is then the user's responsibility to ensure that his USR 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 the (INTEGER) vector NOVATABLE, so you have to trick the FORTRAN compiler into giving you the value stored in NOVATABLE (ICHPTR) as a Floating Point value. You do this using the EQUIVALENCE statement (this is messy, but as far as I know it is the only reasonable way to do it — sorry)

```
        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 - by the compiler) INTEGER*4 value in NOVATABLE (ICHPTR) (which is REALLY a bit pattern representing a floating point number, remember) and try to 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 will not work).

2. Accessing Variables

Variables in NOVA are things that depend explicitly 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 your OPSEQ so that this assumption is no longer valid).

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

LISTPTR = NVEVPTR (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 one of the functions:

IVALUE = NVEVAL (NOVATABLE, LISTPTR)
XVALUE = FPEVAL (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 = NVEVPTR (NOVATABLE, 'CHAMB1_DIST')
DIST = FPEVAL (NOVATABLE, LISTPTR)

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

If the variable is actually a Condition, NVEVAL 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 per event, so there is no danger of incrementing software scalers twice. Thus, if you were to write (for some silly reason):

LISTPTR = NVEVPTR (NOVATABLE, 'CHAMB1_DIST')
DIST1 = FPEVAL (NOVATABLE, LISTPTR)
DIST2 = FPEVAL (NOVATABLE, LISTPTR)
DIST3 = FPEVAL (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. You can, in fact, use NVEVAL (or FPEVAL) for PARAMETERS as well — this is probably the safest method to use unless you are absolutely sure that your PARAMETER is never going to be changed to a variable (direct access of the table is MUCH faster, but less safe).

3. Spectrum Incrementing

The function

ISPECPTR = NVSPECPTR (NOVATABLE, 'name')

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

```
ISX1PTR = NVSPECPTR (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. USREA

As you might expect, routines like NVVALPTR 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 *faster*, 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 which is when things might be moved around), you should be able to determine the positions of such named variables "once and for all". This is where the routine USREA comes into play (in previous versions of the system, this routine was called USRSETUP. For compatibility, this still works — the default USREA function just calls USRSETUP. The default USRSETUP does nothing).

This routine is called by NOVA every time you start analysis (the ANALYSE or EA command), so you can put your calls to NVVALPTR, NVSPECPTR 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 / EA command, USREA will get called again and you will be able to get the new positions of these variables.

So USREA might look something like:[n]

```
      SUBROUTINE USREA (*, NOVATABLE)
      INTEGER*4 NOVATABLE (0:1)
C
      INTEGER*4 ICHPTR, LISTPTR, ISPECPTR
      COMMON /USER_PTRS/ ICHPTR, LISTPTR, ISPECPTR
C
      ICHPTR = NVVALPTR (NOVATABLE, 'CHAMB1_DIST')
      LISTPTR = NVEVPTR (NOVATABLE, 'CHAMB1__DIST')
      ISPECPTR = NVSPECPTR (NOVATABLE, 'SX1')
C
C     USREA 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
C
C     This is an Error Return
C
          RETURN 1
      ELSE
          RETURN
```

[n] Note that earlier versions of NOVA differed in the order of the arguments passed to USREA and USRSETUP.

```
      ENDIF
      END
```

All of these functions (NVVALPTR, NVEVPTR ...) return a *negative value* if the specified variable/ spectrum doesn't exist. USREA could (should) check all of these and take the alternate return (RETURN 1) if your NOVA tables are in error. The system will then issue the error message 'Analysis aborted by User EA routine' — if more detailed information is desired, USREA should write a message to the terminal. 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 USREA 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     The check on pointers being non-negative is not needed if
C     done already in USREA.
C
      IF (ICHPTR .GE. 0) IDIST = NOVATABLE (ICHPTR)

      etc.
```

5. USRDA

As well as the USREA routine (which is called every time you enter the command EA - just before the first event is fetched from the input stream), NOVA also gives you the option of supplying a user routine which is called every time you enter the DA command. The original idea was to allow you to do "end of run" clean up (perhaps calculate statistics for a run etc.). This is not (yet) done in a very clean way, because in fact USRDA doesn't know if it is the end of a run, or just a temporary pause in the middle of a run. One (very Kludgy) way around this (until I do it right) would be for you (the user) to define a variable called something like I_AM_DONE. It is then up to you to set this variable to something (perhaps 1) to indicate that a run is really finished, as a signal to USRDA to perform whatever end-of-run clean up is needed. Like USREA, this routine is passed the entire table structure as an argument, and is expected to take an alternate return if there is an error - i.e.

```
SUBROUTINE USRDA (*, NOVATABLE)
INTEGER*4 NOVATABLE (0:1
C
      IPTR = NVVALPTR (NOVATABLE, 'I_AM_DONE')
      IF (NOVATABLE (IPTR) .NE. 0) THEN
C
C     Do whatever needs doing at end of run
C
            IF (.NOT. OK) THEN
                  RETURN 1
            ENDIF
      RETURN
      END
```

6. Terminal Interface

It is often necessary for one of your User functions to write a message to the terminal screen (e.g. an error or informational message). One of the most blatant inconveniences of earlier versions of NOVA is that such messages always appeared on the terminal screen of the user who started the subprocess. Because of the MAPTO command, this was often <u>NOT</u> the process which issued the command which caused the message. Not only is this annoying to the original process, it is dangerous because the issuer of the command receives no error and therefore assumes that everything is fine.

In this version of the system, such messages can be directed to the terminal who issued the command, but whoever writes the code (me, for system stuff, but YOU for user stuff) has to be a little more careful. Rather than writing to Unit 6 (the normal FORTRAN output unit) or using the TYPE/PRINT statements, all such messages should be written to unit number ITTSUBP (which is defined in the include file 'ncontrol.h'). This unit is re-assigned for every command to be the terminal of the issuer of the command, and so all messages written to this unit appear at the appropriate terminal. There are also three system-supplied subroutines which make use of this.

CALL TTSTR ('String')	Writes 'String' (which may be either a constant string or a CHARACTER variable) to this unit number.
CALL WRREV ('String')	This does the same as TTSTR, but in addition writes the string in reverse video.
CALL TTBELL	Rings the bell on the terminal

So if one of your User routines decides to write something to the screen, it should do one of the following:

1)
```
        INCLUDE 'ncontrol.h'
        ...
        WRITE (ITTSUBP, 100) I
 100    FORMAT (1X, 'The value ', I5, ' is illegal')
```

2)
```
        CALL TTSTR ('You are not allowed to do that now')
```

3)
```
        CALL WRREV ('Error - this function is illegal')
        CALL TTBELL
```

If you write to unit 6 or use the TYPE command, the message will appear on the <u>original</u> terminal which started the NOVA subprocess, which is probably not what you want.

7. Other Functions

This is all that has been implemented as far as 'user-callable' NOVA functions is concerned. I <u>could</u> provide things that let you define new histograms in your USREA 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, let me know and I will consider it.

<u>SCALERS Processing</u>

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 is coming from VDACS (either ONLINE or from a VDACS/NOVA tape or disk file).
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 (apart from an optional fixed-length part at the beginning) this event contains *only scalers* (i.e. the first of these VDACS "Block Read Header words" occurs at some fixed offset - ofetn 0 - in the event record, and everything after that consists of scaler block reads).

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 this routine also assumes that *it will get every scaler event to process.* If it doesn'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 specified as MUSTPROCESS events[o].

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
!               counts/second rather then counts / N seconds).
REAL*4 DELTIME
DELTIME=5.0
!       Number of scalers in the scalers event
NUMSCL = 100
REAL*4 RATES(100), TOTS(100), OLDS(100)
PARAMETER TOTS, OLDS
RATES = USR0 (RATES,TOTS,OLDS,NUMSCL,$RAW,DELTIME)
!       The offset in $RAW of the first scaler block
!       Defaults to 0
$SCLROFFSET = 10
```

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

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

This 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 a program — NSCALERS — that allows you to look at the scalers (nicely formatted) from VAX DCL. Note that if you want to use this program, you *must* use the names given above (at least you must use the names TOTS and RATES — the others are arbitrary).

The $SCLROFFSET variable (this is new) is convenient for skipping over an optional fixed-length part of the event at the beginning (for example, the event / bank headers present in the YBOS event format)

---

[o] A word of warning is in order here. There have been some indications that the MUSTPROCESS statement in TWOTRAN may not work reliably. If you suspect that this is the case, contact someone in the VDACS support group and show them the problem. I have never been able to reliably demonstrate such a fault.

File Handling

It is possible to make NOVA read data files which have been compressed (e.g. using gzip). This can be \useful for saving disk space if you typically analyze from disk rather than directly from tape (this works only on Unix systems).

First of all, you have to create a Unix named pipe (also called a FIFO). These are typically (although not necessarily) created in the /tmp directory. From the Unix shell:

```
mkfifo /tmp/pewg
```

Next, you run whatever program uncompresses your data file and direct the output to this FIFO. For example:

```
gunzip < compressedfile.gz > /tmp/pewg
```

The semantics of this command will be different for other (de)compression programs. The important point is that the uncompressed data is written to the FIFO.

From within NOVA, you then treat the FIFO as a normal inut file

```
Nova> IMODE YBOS (or whatever)
Nova> OPEN /FILE=/tmp/pewg
```

Tape Handling

One of the biggest improvements in this version of NOVA (and also the one which needed the most improving) was the handling of analysis directly from tape. This section discusses the steps required to make the best use of NOVA when reading data directly from tape.

First of all, before you start you NOVA session you should define the input tape unit as a Logical name (this is not so important for VMS, although I think it makes life easier, but it is absolutely essential for Unix because of the / characters needed in pathnames). For VMS

DEFINE MTAPE MUA0:

while for Unix

setenv MTAPE /dev/nrmt0h

(the actual tape unit that you use on your system may be called something different - MKA0 for example. For Unix you should specify the 'nr' form of the tape device — 'n' means it will not be automatically rewound and the 'r' means it is to be accessed in 'raw' mode by NOVA).

For VMS you can (but need not) MOUNT the tape also before you enter NOVA. If you do this, make sure you mount it as a FOREIGN tape (NOVA takes care of label processing by itself).

MOUNT/FOREIGN MUA0:

If you don't do this, NOVA will do it for you the first time that you try to access the tape.

Now you run NOVA. The first NOVA command that you should enter is also a MOUNT command

MOUNT MTAPE

Note that this is independent of whether you did a VMS MOUNT or not. This is a "logical" tape mount - it instructs NOVA to rewind the tape (so that it knows where it is positioned within the tape) and also to read the first thing on the tape to see if it a labelled tape or not. This needs to be done only once. The tape unit "belongs to" (and is managed by) the NOVA subprocess, so popping out of the NOVA command line interpreter (to VMS DCL) and back in does not require another MOUNT command.

The MOUNT command also informs NOVA that all further input data files (things specified in the OPEN/INPUT command) refer to files on the tape rather than disk files. Thus, to open the tape file EXPTxxxxxRUNyyyyy on tape, you simply

MOUNT MTAPE
OPEN/INPUT EXPTxxxxxRUNyyyyy

The MOUNT command tells NOVA to look on the tape for this file rather than on disk. NOVA will now begin searching <u>forwards only</u> on the tape for the requested run. When it finds it, it will begin reading and processing events from it.

If the run that you want is either before your current position on the tape or a long way down the tape, you can use the **REWIND** and **SKPF** commands to position the tape. **REWIND** rewinds the tape to the beginning. **SKPF n** skips runs either forwards (n > 0) or backwards (n < 0). If skipping backwards (n < 0), the <u>last</u> operation is a <u>forward</u> space so that the tape is left positioned immediately after a file mark (and thus the next thing to be read will be "real data").

Since the tape drive is "owned" by the subprocess, things like CTRL-C to abort a tape operation don't work directly, since your terminal is not directly connected to the subprocess. To abort a tape operation in progress (such as searching for a run that isn't there), type CTRL-C and then enter the **MTABORT** command. Be prepared to wait a few minutes.

When you are done with the tape (and, for example, want to analyze from a disk file) you should enter the **DISMOUNT** command. This logically dismounts the tape (note though that the tape is not rewound and/or ejected from the tape drive).

<u>YBOS/NET Interface</u>

There are a couple of special points to be kept in mind when using NOVA to analyze data from a NET connection (either IMODE NET or IMODE YBOS which specifies an online connection).

The PSI package MIDAS specifies each “event” with two (16 bit) numbers - an Event ID and a Trigger Mask. NOVA identifies each event with a single 16 bit number - the Event Type. Additionally, MIDAS allows different Event ID’s to be specified as MUSTPROCESS or MAYPROCESS (similar to the MUSTPROCESS directive in TWOTRAN).

To make use of this, NOVA allows you to define two variables $MUSTMASK and $MAYMASK, which are consulted whenever you open a NET connection (more precisely, they are consulted whenever you EA, so you can change them dynamically). Each of these is treated as a bit-mask, and allows you to specify each Event ID as either MUSTPROCESS (i.e. all events with this ID will be delivered to NOVA - slowing down data acquisition if NOVA cannot keep up) or MAYPROCESS (sent to NOVA only if there is time). Bit 0 (Least Significant) of each of these corresponds to MIDAS Event ID 1, Bit 1 to Event ID 2, etc. If a particular bit is not set in either $MUSTMASK or $MAYMASK, events with this ID are not sent to NOVA at all.

The default for MIDAS is the following groupling of Event ID’s (remember that an Event ID specifies a collection of possibly many different event types with further selection on the basis of Trigger Mask).

ID	Bit Pattern	Meaning
1	1	Periodic Events (e.g. Scalers)
2	2	Physics Triggers (“Real Data” Events)
3	4	Control Events (Start/Stop Run)
4	8	Slow Control
5	16	Calibration

Note though that this table just specifes defaults - your particular MIDAS front-end program might use a completely different scheme.

This division of Event ID’s is used by NOVA ONLY for the MAY/MUST process separation. The NOVA Event Type comes from the MIDAS Trigger Mask. What this means is that although MIDAS allows events with different ID’s to have the same Trigger Mask, you should NOT do this if you will be analyzing the data with NOVA, as NOVA uses ONLY the Trigger Mask to differentiate different event types.

In addition, the following sub-division within Control Events has been adopted.

Trigger Mask	Meaning
‘8001'x	Beginning of Run
‘8002'x	Pause
‘8003'x	Resume
‘8004'x	End of Run

Logical Names

There are a number of VMS Logical Names (Unix Environment variables) which can/should be set for NOVA to function properly. Most of them have been mentioned elsewhere — I include them here to provide in one place a reference list of things that you might want to define.

NOVADIR This defines the directory / Unix path which contains all of the NOVA programs which you might want to run except (possibly) NOVAMAIN.EXE which you might have built yourself with NOVALINK. It MUST be set to something for NOVA to work at all — typically it is defined system-wide to point to the standard NOVA system directory. If you have a special version of NOVA (not your NOVAMAIN.EXE but something else in the system which I have built for you) you should

DEFINE/JOB NOVADIR disk:[directory]

before you execute ANY NOVA commands. Note that the /JOB qualifier is REQUIRED if you redefine NOVADIR in this way (in fact, the /JOB qualifier never hurts in the context that it is used in NOVA, so it is a good idea to use it always). In Unix, you would

setenv NOVADIR pathname

NOVAUSER This defines the user library to use when linking a new version of NOVAMAIN.EXE with the NOVALINK command. Under VMS it typically contains a disk/directory and a library name:

DEFINE/JOB NOVAUSER disk:[directory]yourlibrary.olb

Under Unix, you can (if you wish) specify several libraries and/or directory pathnames — things following -L define pathnames and things following -l specify the name of the library file (excluding the 'lib' at the front of the file name and the '.a' extension). Note that the string should be enclosed in double quotes in this case, as it contains blanks.

setenv NOVAUSER “-L/usr/myname/dir -lmylib -llib2"

NOVASPMEM This number (in decimal) defines the amount of memory to allocate for spectrum storage — it is in units of 1024 channels. Note that such memory is allocated when you issue the NOVASTART command — it cannot be altered (either increased if you need more or released to be used by others) later.

NOVAPRINT This defines the command to be used to make a hardcopy print on a remote printer queue. It should include the string "%f" somewhere, which will be replaced by the name of the (internally generated) disk file which NOVA uses for hard copies.

EDITOR Under Unix, you specify the name of the editor that you want to use (from within NOVA) to service the EDIT/OPSEQ command with this variable. Under VMS there is no choice — the editor is always EDT — but you can customize it with the startup file

DEFINE EDTINI disk:[directory]file.EDT

Batch Operation

NOVA is used most often in an interactive mode (i.e. the User is sitting typing commands at a terminal). It can, however, be used also in a batch mode (I mean by this not necessarily that it is run as a "batch job" on some system "batch queue" - only that the commands which control its operaton come from some disk/command file, not directly from a user sitting at a terminal). Operation of NOVA is essentially unchanged in batch mode, but there are a few differences of which you sghould be aware.

- **Confirmation** In interactive mode, the program often asks you questions in response to non-standard situations (e.g. You have some undefined variables - OK to EA anyway?). In batch mode no questions are asked - the program just carries on as if the answer to the question were Yes (this is why some of the questions asked by NOVA may appear to be "backwards" - they are worded such that the default answer of Yes is what is most often appropriate).

- **NOVABATCHSTATUS** A typical way to run NOVA in Batch Mode is to start up the program, issue whatever commands are necessary to get it going, and then wait for the End of a Run (using the NOVAWAIT program) before DUMPing results and (possibly) starting on another run. This mode of operation will fail if you make a serious error (e.g. specifying a non-existent data file in the OPEN/INPUT command) - things never get started properly, so never reach End-of-File, and NOVAWAIT will wait forever.

  To overcome this problem, the NOVAWAIT command will wait for approximately 10 minutes, during which time it monitors also for the number of analyzed events to increase. If this does not happen, the assumption is that something is wrong, and it returns prematurely, first setting the value of an environment variable (VMS Logical Name) NOVABATCHSTATUS to "STALLED" (note - this is a character string). If all is OK, the variable will be set to the value "OK" (also a character string). Your command file / shell script can (should) check the value of this environment variable and take whatever action is appropriate (terminate the Batch Job, skip to the Next Run etc.) if its value is not "OK" (the only other value currently defined is" STALLED", but more may be added in the future).

- **Unix** Environment variables / user aliases are treated somewhat differently by Unix and VMS. In particular, they are not necessarily inherited by Child Processes. What this means is that (depending on how you define things like NOVADIR), a batch job started from a shell scripot may or may not know about NOVADIR and/or aliases (such as novastart).

  The easiest way to ensure that things work is to include specifically in your Unix shell script lines like the following:

  ```
  setenv NOVADIR ~pewg/v20/nova (or whatever NOVADIR should be defined as on your system)
  source $NOVADIR/novasetup
  ```

  This should ensure that the proper NOVA environment is set up in your batch job.

User Event Driver

If you want to build an input driver for a different event format, it is probably easiest to send me the details and I will write it for you. If this is not possible, I give here a brief description of what is required for you to write your own event driver. This driver will be invoked if you specify IMODE USER.

At a minimum, a user-written input driver must provide a subroutine

SUBROUTINE GEVUSER (*, NOVATABLE, IRAW, NXTPTR)
INTEGER*4 NOVATABLE (0:*)
INTEGER*4 IRAW (0:*)
INTEGER*4 NXTPTR
INCLUDE 'nevinput.h'

This routine is called by NOVA in order to fetch the next "event" from the input data buffer IRAW. It must behave as follows:

1. It should set the variable NXTPTR to the index in the raw data buffer IRAW where the <u>data</u> part of the next event starts (that is, IRAW (NXTPTR) corresponds to the NOVA value $RAW(0). Note that IRAW starts at index 0, not 1.

2. It should set the following variables (defined in a COMMON area in 'nevinput.h') from information in the event header (if it exists)

IEVTYPE	Event Type
IEVLEN	Event length in (16-bit) words.
IEVSEQLO	Low order 16 bits of Event Sequence number
IEVSEQHI	High order 16 bits of Event Sequence number

3. If, when this routine is called, the variable NXTPTR is negative, the raw data vector IRAW is <u>empty</u>. Your user routine should take whatever action is appropriate to get the next buffer of data from tape or disk. It may (but need not) call the NOVA subroutine

   CALL GETBUF (*, ITABLE, IRAW)

   to fill the vector IRAW.

4. Your routine should take the alternate return (RETURN 1) if there are no more events to be analyzed.

If you plan to fill your event buffer by calling the routine GETBUF, then you must also provide a subroutine to do this.

SUBROUTINE GBFUSER (*, NOVATABLE, IRAW)
INTEGER*4 NOVATABLE (0:*)
INTEGER*4 IRAW (0:*)

This routine should read the next buffer of data into the vector IRAW. As with GEVUSER, it should take the alternate return (RETURN 1) if there are no more (end-of-file).

You <u>may</u> also provide the following routines. Default routines (which do nothing) are supplied for those which you do not include.

SUBROUTINE OPUSER (*, ITABLE)

This routine opens the input file (called by the OPEN/INPUT command).

SUBROUTINE CLSUSER (*, ITABLE)

This routine closes the input file (called by the CLOSE/INPUT command).

SUBROUTINE EAUSER (*, ITABLE)

This routine provides the capability of preparing for the EA command. It is called before the routine USREA. The difference is that EAUSER is specific to the user input channel (i.e. it is called only if you have issued the IMODE USER command), while USREA is called for any input device / IMODE.

SUBROUTINE DAUSER (*, ITABLE)

This routine provides the capability of special action for the DA command. Again, it is specific to IMODE USER, while USRDA is called for any device / IMODE.

SUBROUTINE WTUSER (*, ITABLE)

This routine is called in response to the "temporary DA" which NOVA performs automatically when executing commands like LIST. The intent for these commands is that analysis will soon (within a few seconds or less) be resumed automatically.

SUBROUTINE RSUSER (*, ITABLE)

This routine is called in response to a "resume" which NOVA performs automatically following the "temporary DA" mentioned above.

PITFALLS

1. Double Logic

The most obvious pitfall in NOVA (at least the most obvious 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, and
ii) The position of the *INCR* statement in the OPSEQ (i.e. inside IF blocks).

As a simple example which illustrates this problem, suppose you define:

DEF1D/COND=C1/... SX

and then in the OPSEQ

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

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 (after the OPSEQ checks C2 and finds that it is TRUE, it "tries" to increment the spectrum, and first evaluates the condition list to see if it is all OK), and you (the user) have to be clever enough (or rather have a good enough memory) to remember that C1 is in the condition list for SX. There are *TWO* conditions on the spectrum SX, but they appear in very different places (one in the OPSEQ IF statement and the other in the spectrum condition list), and it is all too easy to forget about 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 conditions on spectra have their place, but their place is probably not together! Pick one method of placing condition on spectra, and stick with it.

NOVA helps you out a little bit in this case. While there are two different conditions on this spectrum, there are software scalers associated with each one that can 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 there is a condition list on SX which is causing the problem. In the other scenario (where you are looking at the condition 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 would be 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.

System Independence

Most of you will be using NOVA on only one type of computer, and so will not be so concerned with transportability between machine architectures. For those of you who are, however, I include here a few ‘hints’ (based on my experience in porting NOVA to different machne types).

- **Generic functions**

By this I mean functions where the same function name is used regardless of the type of argument(s). For example, most compilers provide a generic function IAND (I, J), which returns the logical AND of the two arguments I and J *regardless of their type* (INTGEGER*2 or INTEGER*4). What happens in fact is either this function call is replaced by inline code or the compiler substitutes an appropriate function which takes arguments of the given type.

Such generic functions probably should be avoided. Not all compilers/systems provide them (e.g. Linux does not).

I have included (in NOVA) appropriate ‘typed’ functions for these particular ‘bit’ operations - I4AND, I2AND, I4OR, I2OR, I4XOR, I2XOR - use these instead. It does require that you know/remember what kind of variables you are dealing with, but if you are writing a program you should know that anyway.

- **$ in variable names**

It comes as a surprise to some people (it certainly did to me) that a $ is not allowed in a FORTRAN variable name. If you read the FORTRAN specs (FORTRAN-77, at least) it indeed states this. Although most compilers let you get away with it, there are some that adhere strictly to the standard and do not allow $ in variable names.

- **_ in function names**

The underscore, on the other hand, <u>is</u> allowed as a character in FORTRAN variable names. One potential problem, though, is that the underscore is treated somewhat specially (at least by most Unix systems) in that it is often appended to the end of the name of a FORTRAN subroutine/function to distinguish it from a (usually C) system routine with the same name (in fact, the system sticks it on anyway just in case there is a name conflict). That’s why when a Unix loader fails to find some of your subroutines, you get error messages like ‘Unresolved symbol yourprogram_’, with the additional _ on the end of the name.

On some systems (e.g. HPUX), the addition of the trailing underscore is <u>optional</u> - you specify a compiler option (+ppu) to enable this. If you see problems like this, have a look at the man pages for your f77 compiler for the presence of such optional compiler flags.

I have also seen some difficulties when the name of a FORTRAN function (or subroutine) contains an embedded underscore - some systems then append an additional underscore to the name (giving names like your_routine_ _). This is only a problem when you are mixing languages (e.g. calling C routines, which typically do not do this 'add an underscore' thing, from FORTRAN).

If any of these problems crop up, most can probably be cured by simply avoiding the underscore character in subroutine and function names.

- **C**

Speaking of calling C from FORTRAN, this raises a whole new set of problems. First of all, arguments in FORTRAN are passed by reference, and thus must appear in C routines as pointers. This is sometimes a bit inconvenient (especially if you have an existing C routine that you want to include but not edit extensively).

One (very nice) way around this problem is to make use of a package called 'cfortran.h'[p]. This is the technique used within NOVA itself (much of NOVA was rewritten in C to make it portable), and its use is highly recommended. There are still problems with character strings (cfortran.h has the capability of dealing with these as well, but it is a bit confusing and the programmer still has to keep his wits about him). VMS (in particular) handles character strings in a completely different way than most other systems.

- **Case sensitivity**

A second (technical, but still important) issue with mixing C and FORTRAN has to do with case sensitivity of external names (functions and subroutines). Most (but not all) Unix systems treat case as significant in external names by default - the function 'junk' is a different function than 'JUNK' (and also Junk, jUNk, etc.). VMS, however, ignores case. The easiest way to avoid such problems is to ensure that all function / subroutine names are distinct regardless of case.

- **Character / Byte variables**

As mentioned above, CHARACTER variables (in VMS) are very different things than a string of bytes. This means (in particular) that you have to be much more careful in passing arguments to subroutines. The standard trick of using a BYTE (or LOGICAL*1) array to hold a CHARACTER variable does not always work. In addition, passing a single CHARACTER to a routine has to be done with care - a CHARACTER*1 variable is not at all the same as a single BYTE either.

Most compilers provide functions ICHAR (C) and CHAR (I) to convert between the two. Their use is sometimes tedious (and seemingly superfluous), but should lead to portable code.

- **EQUIVALENCE**

Be VERY CAREFUL when using the EQUIVALENCE statement. As noted above in the discussion of the FORTRAN interface to NOVA subroutnes, it is sometimes unavoidable, but its use is strongly discouraged. It is probably safe to EQUIVALENCE two (or more) variables which have the same storage size (e.g. INTEGER*4 and REAL*4), although you still have to be careful how you use them (all possible bit patterns are legal INTEGER*4 values, but some are not legal Floating Point numbers, and may cause your program to crash if used this way). It is almost certainly NOT OK to EQUIVALENCE things of different sizes. In particular, the trick of using EQUIVALENCE to get at the two INTEGER*2 parts of an INTEGER*4 longword is definitely NOT machine transportable - what works on a VAX or DECSTATIONS will not work on an HP machine.

- **Argument types**

[p] This package was developed by Burkhard Burow at DESY, and is available via anonymous ftp from zebra.desy.de.

Argument types must agree when calling subroutines. Many of us (me included) tend to get lazy about this, particulary since (on DEC machines at least), you can pass an INTEGER*4 value to a subroutine and have it access the passed value as INTEGER*2 (or BYTE) with no difficulties. This most definitely does not work on machines whose architecture has the bytes the other way aroung (e.g. SGI and HP). Note also that INTEGER constants are almost always passed using the natural word size of the machine (INTEGER*4 usually). So if you have a subroutine which expects an INTEGER*2 argument, a call like

```
        CALL MYROUTINE (1)
```

is almost certainly going to cause problems - the constant ‘1' will be treated by the compiler as an INTEGER*4 value. One (tedious, but also useful in other ways) way around this is to declare ALL such constants as PARAMETERS. The code

```
INTEGER*2 ONE
PARAMETER (ONE = 1)
...
CALL MYROUTINE (ONE)
```

should work regardless of machine architecture.

Index