UNIVERSITY OF CALIFORNIA
LICK OBSERVATORY TECHNICAL REPORTS
NO. 34

THE VISTA PROGRAMMER'S GUIDE

Tod Lauer
Richard Stover
Donald Terndrup

Version 2
Santa Cruz, California
May 15, 1984

I. Overview

The VISTA *Programmers' Guide* explains the basic operation of the program VISTA, the philosophy that guided its development, and the procedure for adding new subroutines. It presents a list of common utilities used by the various subprograms, and a set of instructions for tailoring VISTA to your own needs.

The beginner to VISTA programming has the task of learning the operation of the program from a user's point of view, then learning the intricacies of the code itself. We realize this requires some effort (one cannot learn all about it in a day), but we believe that it is not too difficult for the average astronomer learned in FORTRAN and experienced with VISTA as a user. This *Guide* will, we hope, ease the first steps toward a thorough knowledge of the program.

II. Philosophy

VISTA is an interactive system of routines for the manipulation of two-dimensional image data. It handles reduction, processing, and display of images, surface photometry, stellar photometry, and basic reduction of images used for spectroscopy. It also provides features to make the program easy to use, and to handle complex or repetative problems.

VISTA was written over a long period of time, during which our understanding of CCD's and our own research interests grew and changed. We also desired that the program be "public," so that (1) other astronomers within the observatory could reduce their CCD data, and (2) the program could expand or change with the needs of its users. Therefore, we have tried to make the routines as versatile as possible, and to make the main program easily modifiable. New routines can be added to the program at any time, with a minimum of effort. It is clear, however, that individuals will need software to deal with research problems which cannot and should not be included in a system for general use. Therefore, the program contains routines to write reduced images, spectra, or other types of data to diskfiles which can then be read by other routines. It cannot be arbitrarily expanded to handle everyone's needs. Future expansion should be limited to general-purpose routines for the processing and analysis of 2D direct images, and to the operations needed to feed reduced data to other software systems.

The main program is a simple and fairly general command interpreter that is independent of most of the details of the subroutines. Those who contemplate new software systems for analysis of 1D spectra, reduction of echelle images, or other complex tasks are encouraged to consider using the main routines of VISTA

as the foundation upon which to build these new systems. We have found that it is easy to build up new systems of software by simply supplying a new set of subroutines for the program to run.

To futher define principles of operation, we identify some of VISTA's basic features.

1. VISTA is a *command-driven* system, which requires the user to enter commands which are interpreted by the program. *Menu systems*, on the other hand, present lists of available programs, asking the user to select from this list the ones desired. In VISTA it is the user's responsibility to know what to do and how to do it. This, of course, is how VAX/VMS operates. It has the advantage that it permits the greatest flexibility and does not "shoehorn" the user into a strict or tedious operation procedure. The disadvantage is that it requires a somewhat greater effort from the user to learn the operation of the program. (The *VISTA User's Guide* gives instructions.)

   Available routines are called up by their command names. The user specifies any additional information needed to modify or control the command with keywords. Routines ask questions only when vital information is missing from the command line. The user is not pestered with excessive queries, selection of options, etc. The commands work cleanly with as little interaction as possible, and have sensible default responses for missing non-essential information. In some cases, of course, interactive commands are desirable; they are limited to those that really need run-time decisions. If possible, interactive input is kept as a keyword activated function. In the routines using an extensive input list, that list is in the form of a disk file whose name can be specified by a keyword. If a task is too complex to be easily accomplished by a single routine, it is broken into several routines.

   VISTA attempts to minimize the protocol needed to operate commands. Most keywords can be entered in any order, and reasonable default values are given for unspecified keywords. The program can also run VAX DCL commands, allowing the user to manage data files or even edit them during a reduction session.

2. The flexibility of VISTA is increased by its *procedures*, which are lists of commands to be executed as a program plus special commands which control the flow of the procedure. These are similar to DCL command files. The program contains a command language complete with control flow, condition testing, manipulation of symbols, user-defined subroutines, symbolic image buffers, and data input. Procedures allow the user to configure VISTA to handle complex or repetative tasks. This helps to eliminate tedium and to reduce the need to add new software to the program.

3. VISTA is "user friendly." All routines are designed to protect the operator from basic errors and issue simple but informative messages when non-recoverable mistakes occur. The routines are written to anticipate or survive runtime software errors, such as variable overflow, division by zero, or read/write errors in files. A session with VISTA can go on for several hours, so an error in the software that causes the program to crash can result in the loss of a significant amount of labor.
   VISTA supports a detailed *helpfile* system. The user can request information on a command or subject while running the program, or print out the whole helpfile as a user's manual. The format of the helpfile is discussed below.
4. VISTA is a *top-down* software system, meaning that the data reduction routines receive most of their input from above, and do not heavily depend on an extensive set of low level utilities. This gives the great advantage that the subroutines can be easily inserted in or removed from the larger program.
   The VISTA software uses all the features available in DIGITAL's FORTRAN-77, which allows well-structured and legible code, largely free of the obscure and confusing constructions often required by older versions of that language. The code is also well documented, and (it is hoped) can be easily understood by someone needing to dig into VISTA to add new routines or to examine a program's operation. We have found that one can do a better job with a data reduction system if one is working with a clear understanding of how it functions. It is *imperative* that new code be clearly documented. Routines that operate as a "black box" cause confusion and uncertainty when they need to be changed.

III. How VISTA Works

i) The top level

The main program is [CCDEV.BASE]VISTA.FOR. It handles all the communciation with the user and with the subroutines. It works as follows:

1. The program initializes several variables and logicals which control its flow.
2. The subroutine INITIAL is called, which defines which directories the program reads from or writes to.
3. A command is received, either from the user's terminal or from the procedure buffer. The logical variable GO is set to .FALSE.

4. The command line is separated into the command word, other character strings, integers, and real numbers. These pieces are loaded into common blocks which the subroutines can read. The commons are declared in the file VISTALINK.INC, which will be discussed later.
5. The program then reads through the *command tree,* which specifies the number of images or spectra needed for each command, their sizes, and the name of the subroutine to which the program will later branch. The branching is not done if GO is .FALSE. (The command tree will be described in more detail below.)
6. When finished looking in the command tree, VISTA.FOR attempts to find the required images and spectra. If they are not there, an error condition results, and the program returns to step 3. If all is well, the variable GO is set .TRUE.
7. The program makes a *second* pass through the command tree. When it enounters the current command in the tree, it calls the subroutine listed there, passing as arguments the desired images or spectra, and their sizes.

ii) How images are stored in VISTA

VISTA uses dynamic memory allocation, which permits users to connect images of any size. The memory needed to store an image or a spectrum is not reserved at compilation time with a DIMENSION statement or its equivalent. Rather, it is requested from the VMS operating system at run-time with a call to a VAX library function in the subroutine CREATEIM. Section iv) discusses the *command tree,* where most of the image handling details are treated, and describes the manner in which subroutines refer to images will be discussed at length later.

iii) Command parsing

The command line is parsed by repeated calls to subroutine DISSECT. This subroutine breaks the line into individual words, examines those words, and returns their values, whether they be integer constants (a number with no decimal point or the values of certain variables), floating point constants (containing a decimal point), or alpha-numeric character strings. the various components of the command line are stored in common blocks defined in VISTALINK.INC. The

first word of the command line is assumed to be the command name, and is stored in the character string COM. The values of integer words are stored in the array IBUF, that of reals in the array CONST, and alpha-numeric words in the chatracter array WORD. Each of these arrays has fifteen elements, so VISTA can accept at most fifteen of each type of word on the command line. Subroutines which are called from the VISTA command tree can examine and use the values stored in IBUF, CONST, and WORD, which are delared in the file VISTALINK.INC. This file is shown in Appendix 2.

iv) The command tree

The command tree is a long series of IF, ELSE IF, ELSE IF, ..., END IF statements, which test the command entered from the terminal or from the procedure buffer. In each IF-block, the values of NEEDIM and NEEDSP are set: they specify the number of images or spectra, respectively, that are passed to the subroutine. The values of these variables are set to zero at the top of the tree, so they do not need to be reset if no images or spectra are being sent to a subroutine.

An exerpt from the command tree is shown as Appendix 1. The commands are not actually in this order in VISTA.FOR; we have rearranged them for illustration. The rules for calling images are as follows:

1. If no images or spectra are required by the command, the IF-block contains only a call to that subroutine. The subroutines are designed to return immediately if the variable GO is .FALSE. This is illustrated in Appendix 1 by the BOX command.
2. If images or spectra are required, the call to the subroutine is preceded by IF (GO).
3. The call to a subroutine can pass either one or two images. The location (memory address) of the beginning of the first image is stored in LOCIM, and that of the second is LOCB. The first image has NROW rows and NCOL columns; the second has JROW rows and JCOL columns. In Appendix 1, the call to ARITHCON passes one image, while the call to ARITH2IM passes two images.
4. One of the images passed to the subroutine may be already stored in the AED television system. The location of this image is the value of LOCTV. The image has NRTV rows and NCTV columns.
5. If only the television image is sent to the subroutine, NEEDIM is set to -1. If the television image *and* another image is being sent, NEEDIM is set to -2. The other image is located at the value of LOCIM. In Appendix 1, the first

case is illustrated by the call to PSF, while the second is shown in the call to FITSTAR.

6. The call to a subroutine can pass either one or two spectra. The location of the first spectrum is stored in LOCS, and that of the second is LOCSB. The first spectrum has NSCOL columns; the second has JSCOL columns. The call to MASH (see Appendix 1) passes one spectrum; the call to ARITH2SP passes two. If the call to a function has both image and spectrum specifiers in the command line, all the spectrum specifiers must come before any of the image specifiers. The variable IMSP tells how many spectrum specifiers there are in the command line.

The command tree in VISTA.FOR is broken into several INCLUDE calls to other files that hold part of the tree. This allows blocks of related programs (spectroscopy, stellar photometry) to be included or excluded in VISTA as desired. These files have extensions '.PGM' in the directory [CCDEV.BASE].

IV. Writing new subroutines

i) Is a new routine needed?

VISTA is designed so that users can add new programs easily, tailoring the program to individual needs. It is important, however, that new programs not be added at such a pace that VISTA becomes unmanageably large, or added without regard to programs already there. Every consideration should be given to ways of accomplishing one's tasks that do not require the writing of a new subroutine. Specifically:

1. Can the task be done with VISTA procedures? The procedures were designed to be as flexible as possible, with conditional branching, loops, etc. Also recall that DCL commands can be executed directly from VISTA, thus adding the power of VMS to that of VISTA.
2. Can an existing program be modified by adding a new keyword to the command? It may be that a new operation is merely a variation on another routine. Those variations are easily handled with keywords.
3. Is the new task so similar (either in concepts or actual algorithms) to an existing subroutine that, although it will be run with a new command, the new code can be included in an existing subroutine? That subroutine can then be modified to test which command called it, branching accordingly.
4. Can the new program make use of the generic calls GET, SAVE, PRINT or PLOT? If so, these programs can be modified by using new keywords in these suboroutines.

5. Finally, and most importantly, is the new task of sufficiently narrow interest that no other user is likely to use it? If it is, then it can either be run in a private version of VISTA, or it can be created as a separate program altogether, using as input the files created by VISTA.

ii) Programming style

The subroutines in VISTA have all been written in a style which we hope makes the programs readable. The importance of this for a "public" program cannot be overstated, for the usefullness of the program depends as much on its readability as on its performance. We *insist* that programs written for VISTA be written in this style to preserve the readability of the program.

1. Use comments extensively! The program should explain itself to the reader. We have found that printing the FORTRAN in upper case, and the comments in capitals and lower case helps one to read the program better. Use "in-line" comments where necessary.
2. Use logical variables to denote conditions. Do NOT use the value of integers or reals for this purpose, if you can avoid it.
3. Avoid the statement GOTO and branching to numbered lines.
4. Use the form DO ... END DO (rather than DO with a number) to delimit DO-loops. Indent the code inside each loop. If there are several levels of nested DO-loops, indent them to show these levels.
5. Use IF-blocks (IF ... END IF) to delimit sections of code exectued under some condition. VISTA programs are often a series of IF-blocks which test logical variables. Avoid IF-tests which branch to numbered lines.
6. Label all variables with a comment describing their use. This should be done the first time a variable is mentioned in the code, and especially for the variables in common blocks.
7. Remember that the VAX saves the values of variables in a subroutine even after the program returns from that subroutine. Repeated calls to the subroutine, therefore, can avoid some repetative calculations. Common blocks are to be avoided, except when it is necessary to transfer values from one routine to another. It is *never* necessary to create a common block to preserve constants in a subroutine for future calls to that routine. Since values are saved after calls to the subroutines, it is important that variables be initialized properly so that "leftover" values from previous calls to a subroutine do not contaminate new calculations. The values used in initializations should be clearly stated in the program.

8. Do not type FORTRAN lines in the old "keypunch" mode, which has the code beginning in the 7th column, having columns 1-6 containing line numbers and with the 6th column holding continuation characters. Use the "screen" mode: (a) the 'tab' character is used before every executable statement; (b) numbered lines have the tab character following the number; (c) the continuation character is tab followed by 0 - 9, then followed by any number of blanks or tabs, after which is the rest of the code.
9. Make good use of "white space" (i.e., blanks and tabs). FORTRAN ignores *all* space or tab characters. Use blank lines to separate the code into convenient, easily understood sections. Use blank characters to separate variable names from arithmetic operators.
10. Use variable names of sufficient length and descriptiveness so that the use of each variable is reasonably clear from its name. VAX FORTRAN supports names up to 32 characters in length. Use the underscore character to make names clear.
11. The beginning of each program has the following format: (a) the first line is the subroutine call, (b) the second line is blank, (c) the third line is a one-line description of the program, (d) the fourth line is blank, and (e) there follows a *complete* description of the program and how it works. At the end of the description, include the name of the author and the date of the program's last revision.

These standards, if carefully followed, will produce legible code in a style consistent with current practice, but will leave room for variations in individual programmer's style and taste.

iii) Interpreting commands

As mentioned above, the main program delivers only images or spectra with the call to the subroutine. The other information necessary to interpret a command is contained in the common blocks in 'VISTALINK.INC', which is listed as Appendix 2. Study this appendix before reading the discussion below.

The first thing any program does is check the state of the logical variable GO with the line

```
IF (.NOT. GO) RETURN
```

Then, the contents of WORD, CONST, etc., are checked to find the conditions under which the program is being run. The appropriate logical variables are set, and then the program begins its execution. Image and spectrum specifiers are loaded into the IBUF array before execution of the subroutine begins.

There are several keywords which should be the same in all programs:

BOX	tells the subroutine to use only a section of the image.
INT	run this subroutine interactively (i.e., the program asks for information rather than looking it up somewhere).
HARD	prints plots on the hardcopy device.
R=n	selects row n.
C=n	selects column n.
S=n	selects spectrum n in those programs (like PLOT) which can operate both on images and spectra.

It is customary, when testing the contents of WORD, to avoid any processing of the options within the IF-block itself. Rather, the tests merely set logical variables or error conditions which are later tested. Besides clearly separating testing and processing, this postponement may speed the subroutine's operation by beginning computations only if all required information is present.

iv) Referring to images

FORTRAN passes data to subroutines in a manner termed *call by reference*, which simply means that the *address* of a variable or of the first entry in array is sent to the subroutine, rather than the values of those quantities. When a image is needed by a subroutine, the address of the beginning of the array is loaded as the value of an integer. When passed to a subroutine, the *value* of that integer is required. The value (instead of the address) can be passed by the %VAL( ) construction in the call top a subroutine. This address assigned to a variable name in the subroutine in the usual way: by placing an array name in the same location in the SUBROUTINE statement as the address is in the call to that subroutine. For example, the command AI, which adds two images, is executed by a call to

```
CALL ARITH2IM(%VAL(LOCIM),NROW,NCOL,%VAL(LOCB),JROW,JCOL)
```

Here the values of LOCIM and LOCB are the addresses of the two images which are being added. NROW and NCOL are the number of rows and columns in the first image; JROW and JCOL are the number of rows and columns in the second image. The subroutine treates them as arrays (here called A and B):

```
SUBROUTINE ARITH2IM(A,NROWA,NCOLA,B,NROWB,NCOLB)
DIMENSION A(NCOLA,NROWA)
DIMENSION B(NCOLB,NROWB)
```

Note that the number of columns is the *first* dimension of the array in the subroutine.

Information about the properties of an image is stored in common blocks defined in IMAGELINK.INC, which is reproduced as Appendix 3. Study this appendix before reading on.

There are two ways subroutines refer to pixels in an image. In the first scheme, all references are to the location in the FORTRAN array that holds the image. The second scheme has the program refer to pixels by CCD image-row and -column number. These are different because, whereas the FORTRAN arrays *always* run from ARRAY(1,1) up to ARRAY(NCOL,NROW), where NCOL and NROW are the number of columns and rows, respectively, the CCD image-row and -column numbers have arbitrary origins. Should a DO loop have index running over row or column number, it is necessary to translate from the DO-indices to the array-indices. This is accomplished using the integer variables ISR and ISC, which hold the starting row number and column number, respectively, for the first pixel in the image array. These parameters are found in IMAGELINK.INC.

v) The work array

VISTA provides a scratch array of size 256^2 REAL*4's to handle temporary storage of variables. The common block is named WORK. It is prefered that you use the common block to store intermediate values, rather than creating new arrays. The common block is used by many VISTA programs, so it will have garbage in it which must be cleared out at the beginning of any program that uses WORK.

vi) Missing information and other errors

VISTA commands should contain all the information a subroutine needs to have to run properly. It is quite common, though, for a user to forget an important keyword, or to make typing errors in the command. Therefore, *all* programs must have some way of detecting missing information and either notifying the user of this lack or requesting the desired stuff.

One of the few inconsistencies among VISTA subroutines is the handling of missing information. Some routines print a message spelling out what is needed to operate the program then immediately return. Other routines print a message asking for the missing quantity, then have a READ statement to receive this. We have generally left it up to the programmer to decide how to handle these situations. Two considerations, however, are of great importance: (1) Any messages

requesting information or informing the user of an error must be self-explanatory. (2) A request for information from the terminal must be general enough so that garbage or erroneous input is detected.

Extending this last item to a broader principle, we remind the programmer that *run-time errors in a program must not cause* VISTA *to crash!* Two common errors and their fixes are:

1. An attempt to open a file that does not exist can be corrected for by using the IOSTAT keyword in the FORTRAN OPEN statement.
2. A read-error in a diskfile or from the terminal can be fixed by using the ERR keyword in the READ statement.

VISTA subprograms usually return as soon as errors are detected. To signal that the command was not completed successfully, the logical variable XERR is set .TRUE.

Some programs can be halted when control-C is typed on the terminal. VISTA sets control-C to run an asynchonous process that sets the variable NOGO to .TRUE. The state of NOGO can be tested at any point in the program. (Note that NOGO is different from .NOT. GO!).

vii) The output and error channels

VISTA provides an output-redirection mechanism, whereby data written to FORTRAN unit 44 can be sent (1) to the user's terminal, (2) to a file, or (3) to the lineprinter. The syntax for this redirection is given on the command line and is handled by the main program. The subroutines do not see this redirection. Use unit 44 for long output to the terminal (tables, etc.) which the user might want saved.

Error messages should always be sent directly to the terminal with a 'TYPE *,...' statement.

viii) Other considerations

When writing a VISTA program, as when writing any computer program on any machine, one must take care to minimize the memory required by the program, and to reduce the run-time by proper use of loops and subroutine calls. There are also considerations unique to FORTRAN on the VAX:

1. Construct DO-loops so that the first index of a multi-indexed array varies fastest.

2. IF-blocks run faster than IF-statements followed by GOTO. With proper use of IF, ELSE, ELSE IF, END IF, the GOTO statement has virtually disappeared from VISTA programs.
3. Use PARAMETER statements to define constants.

A longer discussion of these matters can be found in the VAX FORTRAN manuals.

ix) A sample VISTA program

Appendix 4 shows a sample VISTA program, ARITH2IM, which is used in arithmetic between two images. It illustrates many fatures of VISTA programs: (1) This one subroutine is called by several command names. (2) Each command has options that may be turned on by the user. (3) The control-flow is handled by logical variables and IF-blocks. (4) The program can handle arbitrary image sizes.

V. How to add a subroutine to VISTA

Parallel with the instructions below are a set of examples. In these examples, we show how to add the program ARITH2IM to VISTA.

i) Adding a new subroutine

1. Write the subroutine. It will be helpful if you debug it as much as possible before trying to add it to VISTA, perhaps by running key sections of the routine as a separate program. Make the name of the file holding the program the same as the name of the subroutine itself, if possible. For example, the subroutine ARITH2IM is found in ARITH2IM.FOR
2. IMPORTANT: Check the two libraries that compose VISTA to see that these libraries do not already contain a module with the same name as the program you are adding. The two libraries are CCDVIST in [CCDEV.BASE] and UTIL in [CCDEV.BASE.UTILITY]. Use the commands 'LIBR/LIST libraryname' to get a list of the modules in these libraries. In our example, we would execute the commands

```
LIBR/LIST [CCDEV.BASE]CCDVIST
LIBR/LIST [CCDEV.BASE.UTILITY]UTIL
```

looking to see that the module ARITH2IM does not already exist in *either* of the two libraries.

2.1 If the module you propose to add is not named in the libraries, proceed to step 3.

2.2 If the module you propose to add is named in either library, you must rename the subroutine call to a name not found with LIBR/LIST. Following step 1, you should also rename the fortran file.

3. Decide which library your subroutine will be put.

   3.1 If the subroutine is called directly by a command, the subroutine will be added to CCDVIST.

   3.2 Is the program a "utility," i.e., does it perform a function of general use? If it is, then it will be added to UTIL.

   3.3 Otherwise, the program will be added to CCDVIST.

4. Compile the program using the FORTRAN command. In our example, we would type

   FORTRAN ARITH2IM

   4.1 If the program does not compile successfully, fix the problems in it, and go to step 4.

   4.2 If the program compiles properly, continue to step 5.

5. Add the module you have just created to the desired library.

   5.1 If adding to CCDVIST, type

   LIBRARY/INSERT [CCDEV.BASE]CCDVIST filename

   In our example, this is

   LIBRARY/INSERT [CCDEV.BASE]CCDVIST ARITH2IM

   5.2 If adding to UTIL, type

   LIBRARY/INSERT [CCDEV.BASE.UTILITY]UTIL filename

   In our example, this is

   LIBRARY/INSERT [CCDEV.BASE.UTILITY]UTIL ARITH2IM

6. Add the call to the subroutine to the command tree.

   6.1 Determine which file of the several that compose the tree will hold the subroutine call.

   6.2 Add an appropriate place in the command tree, add

```
ELSE IF (COM .EQ. 'somestring') THEN
    NEEDIM = somenumber
    IF (GO) CALL SUBROUTINENAME
```

In our example, we write

```
ELSE IF (COM .EQ. 'AI' .OR. COM .EQ. 'SI') THEN
    NEEDIM = 2
    IF (GO) CALL ARITH2IM(%VAL(LOCIM), ... )
```

7. Link VISTA. Execute the command

   @[CCDEV.BASE]LINKVISTA
8. Debug your program in the VISTA system.
   8.1 If it does not work, return to step 1.
   8.2 If it seems to work, make sure that it works under all circumstances. Try omitting keywords, running it with varying image sizes and origins, etc.
9. Find the helpfile [CCD.HELP]HELPFILE.LIS. Add a new entry according to the pattern of entries already there. Place your new entry near those for programs that have similar operation. Make your entry a *complete* set of instructions for the user. Run the program [CCDEV.BASE]MAKEHELP which processes HELPFILE.LIS into smaller helpfiles which the VISTA command HELP reads. (This takes about 4 minutes.) Clean out old versions of the helpfiles by giving the command PURGE [CCD.HELP].

ii) Modifying an existing subroutine

Here we again use ARITH2IM as an example.

1. Make the appropriate changes to the subroutine. *Keep the old version of the program* in case you cannot get things to work.
2. Find which library holds the object code for the program you are working on.
   2.1 If the program is found in [CCDEV.BASE], the module will *probably* be found in the library CCDVIST in that directory. In our example, if we are modifying ARITH2IM in [CCDEV.BASE], we can expect ARITH2IM to be found in CCDVIST.
   2.2 If the program is found in [CCDEV.BASE.UTILITY], the module will *probably* be found in the library UTIL in that directory.
   2.3 Confirm the location of the module by typing

LIBR/LIST [CCDEV.BASE]CCDVIST
LIBR/LIST [CCDEV.BASE.UTILITY]UTIL

and examining the output for ARITH2IM.

3. Compile the program with the FORTRAN command. In our example, we type

   FORTRAN ARITH2IM

   If all worked, proceed to the next step. If not, return to step 1.

4. Replace the module in the proper library with the object file you have just created with FORTRAN.

   4.1 If the program belongs in CCDVIST, give the command

   LIBRARY/REPLACE CCDVIST filename

   In our example, this is

   LIBRARY/REPLACE CCDVIST ARITH2IM

   4.2 If the progam belongs in UTIL, give the command

   LIBRARY/REPLACE UTIL filename

5. Modify the call to the subroutine, if necessary.
6. Link VISTA by executing the command

   @[CCDEV.BASE]LINKVISTA

7. Debug the command in the VISTA environment. Follow the cautions listed above.

8. If the instructions for operating the new version of the subroutine are different from old instructions, modify the appropriate entry in the helpfile.

VI. The VISTA helpfile

It is extremely important that each command have complete operating instructions in the helpfile. The instructions should be up to date, reflecting the current operation of the command. No addition or modification to VISTA is complete until the corresponding changes are made in the instructions.

Appendix 5 shows a section from the helpfile containing instructions for the operation of the subroutine ARITH2IM. When making changes to the helpfile, follow the pattern displayed in this appendix.

The HELP program reads the file line by line. If the line begins with one of the special characters :, ., {, or /, the program interprets the line in a special way. Otherwise, the line is assumed to be instructions, and is displayed to the user as it is written in the file.

The symbol : designates a *subject*. The VISTA commands are arranged according to these subjects. Examples are "Spectroscopy," "Stellar Photometry," etc. The line following a subject definition must be a command definition.

Lines beginning with '.' denote *commands* or *major topics*. The period is immediately followed by one word (which is the topic), a single space, and several words that make up a title for that command. The user can receive information on a command or major topic by issuing the VISTA command 'HELP topic'. Several commands can share the same text. The text in Appendix 5 is displayed for the commands 'HELP AI', 'HELP SI', 'HELP DI', and 'HELP MI' .

Following the command definition are several labels which are used only in the production of the index for a complete listing of the helpfile. '/' denotes an entry in the index, and '//' a sub-entry that will appear under the first. The index entry and sub-entry must appear on the same line. Either can have more than one word.

The line $PAUSE halts output directed to the terminal until the user hits RETURN. This breaks the text into convenient screenfuls. A screenful of text should contain no more than 20 lines, so that a message can appear at the bottom of the screen.

Lines that begin with { are printed only when the output of the HELP program is sent to the user's terminal. If the information is being directed to a file or sent to the lineprinter, these lines to not appear. Make sure that a screenful of text delimited by $PAUSE counts these lines!

VII. VISTA'S programming utilities

Since VISTA is a top-down system, its high level routines do not rely heavily on a set of basic subroutines to perform their tasks. Still, there are a number of useful utilities which are summarized below. For more information you should consult the the source code. These should be examined by those writing new routines. Many of the these utilities, however, are only used by the top structure of VISTA and some special purpose routines, and need not concern the average programmer.

i) Textstring, keyword, and parsing utilities

These utilities handle much of the work of the VISTA command interpreter, and hence are responsible for much of the "feel" of the system. They permit the user to talk to the routines in a flexible format, and handle the input and output of text strings. The most important utility is ASSIGN, which handles the task of interpreting keyword assignments, and is therefore used by most of the high level routines. FILEDEF is used by routines which work directly with disk files. VARIABLE allows routines to pass back scalar results to the user level.

ASSIGN — is used to parse keywords of the form KEYWORD=EXPRESSION, returning the numerical value of the expression. Since the user routinely specifies parameters to the subroutines with keywords of this form, ASSIGN is called very often.

ASSIGNV — works just like ASSIGN, except that it will calculate the values of a list of expressions assigned to a keyword. This permits the user to pass several values to a routine with one keyword. For an example of ASSIGNV, see code for the MASH command, which can be found in MASH.FOR in the directory [CCDEV.BASE]

DISSECT — is the parser used by VISTA to break a line of text into seperate alphanumeric words, integers, and floating numbers. It is used heavily by the VISTA interpreter and by other routines which must parse complex strings of text.

FILEDEF — is used by all routines which use diskfiles. It takes file names given and tacks on default directories and extensions if they are missing from the supplied name.

SUBVAR — loads or reads subscripted variables from VISTA's variable table. The subscript is actualy a number tacked onto the variable name. See the code for MN command, located in AVERAGE.FOR in [CCDEV.BASE], for an example of its use. See the entry below on the VARIABLE routine for more information.

UPPER — is a function which converts character strings to upper case and returns the number of characters in the string. This routine allows VISTA to process lower and upper case entries in the same way. UPPER is an INTEGER FUNCTION, so don't foget to declare UPPER as in integer in a routine which calls it.

VALUE — is the utility used to parse algebraic statements involving

variables. It is used mainly by ASSIGN and ASSIGNV, but can be used in any routine where an algebraic string is entered by itself, rather than as the right side of a keyword equality.

VARIABLE
: permits the user to define variables with the SET and ASK commands, and for routines to load scalar results into variables for access by the user or other routines.

ii) Image and spectrum array utilities

VISTA uses these utilities to get virtual memory from the system, pass the addresses of image and spectra arrays to the high level routines, and to rearrange portions of the arrays. The most important utilities are GETBOX and MASK. The former allows routines to process a subsection of an image, while the latter enables routines to ignore parts of an image. For the most part, the other utilities are not commonly used by subroutines, but are used by VISTA itself to get memory for new images or spectra, and to connect the memory to the high level routines. These utilities will be needed, however, in routines which generate new images or spectra from old ones, handle more than two images or spectra at once or which only process images or spectra as an option selected within the routine itself.

COPIO
: copies one 2D array to another with a possible offset. It is intended for use in routines which only have access to the virtual addresses of the image buffers, rather than the FORTRAN array elements which hold the image data.

CREATEIM
: is used by VISTA to setup and get the virtual memory needed for new image buffers. It should only be used in routines which create new images. CREATEIM looks for the image specifier in the IBUF array; it needs to know which image specifier to use (for example: create an image using the second image specifier on the command line.) It then calculates the amount of memory needed for the image array and attempts to get it from the system. If successful, it loads the parameters of the new image into the common blocks held in the IMAGELINK include file.

CREATESP
: works like CREATEIM, but defines new spectrum buffers.

GETBOX
: is used by all routines which have the option of processing a subset of an image within a user-defined BOX rather than the whole image. The GETBOX utility compares the size

and origin of the image to that of the desired box, and returns the origin and number of the rows and columns of the image to be processed.

GETIMAGE is used by VISTA to pass the addresses of the image arrays to the high level routines. It also loads a list of parameters describing the images into a common block for easy access by the routines. GETIMAGE looks on the command line to find which image buffers to pass to the routine. GETIMAGE should only be used in the cases where VISTA cannot pass the addresses to the routine, such as routines that must operate on three or more images at once, or routines that only process images as an option (such as the PRINT and PLOT commands).

GETSPEC works just like GETIMAGE, but for spectra.

MASK tests if a given pixel within an image is "masked out," (i.e., to be ignored in a calculation).

SPECTRANS works like COPIO, but for spectral arrays.

iii) FITS header routines

VISTA uses FITS headers to contain the information and parameters associated with images, spectra, and other data. A FITS header is a long character string which contains a set of keywords to identify various parameters, such as the size of an image, its label, date of observation, etc., and the values of the parameters. VISTA uses the utilities below to look up, set, or print out the values of the various parameters stored in the header. For more information, see a FITS manual.

GETHEAD is a package of three functions, IHEAD, FHEAD, and CHEAD which look up and return integer, floating, and character parameters, respectively, held in a FITS header. Note that VISTA already provides the high level routines with the sizes and origins of the images and spectra.

GETHJD reads the FITS header and calculates various parameters of the observation, including the heliocentric Julian date, the air mass, etc. All routines which need such information should use GETHJD.

HEADER prints out a formatted image header.

HEADSET	is a package of three utilities, INHEADSET, FHEADSET, and CHEADSET to load integer, floating and character parameters, respectively, into a FITS header.
SPHEADER	works like HEADER, but for spectra.

iv) Mathematical utilities

There is no standard mathematical library that VISTA uses, nor is there a well defined set of routines that handle the computational needs of the higher level reduction routines. There are, however, a few common problems which are handled by these utilities:

BIN	contains a number of routines for doing random-access 2D image interpolation. The different routines trade off various degrees of speed for numerical accuracy.
FINDCENT	finds the centroid of an object in an image.
GNLLS	is a general routine for handling non-linear least squares fitting of functions.
MEDIAN	uses a quick-sort algorithm to find the median of a 1D array.
SINCGEN	is used for high-accuracy 1D interpolation.
SOLVE	solves a linear system of equations.
SPLINE	produces a 1D spline on a set of points.

v) Graphics

VISTA uses the MONGO package for its graphics, There are, however, a few useful utilities:

IAMVT100	finds out if the user is on a VT100 terminal. It should be used by all graphics routines to avoid sending graphics commands to non-graphics terminals.
VTG100	is a complete package of low-level VT100 graphics routines written by Richard Stover. It contains routines for vector and text plotting, and interactive routines using the VT100's internal cursor.

vi) Miscellaneous

MATHCRASH	enables routines to recover from mathematical errors which would otherwise cause VISTA to crash.
WAIT	tells the VAX to put VISTA in hibernation for a specified period of time. It is used to avoid tying up the CPU by routines running in loops waiting for asynchronous input.

VII. I/O routines

We list here the subroutines used by VISTA for input and output. Knowing the details of the file and tape format for images or spectra may be helpful when using VISTA in conjunction with otther programs. For example:

1. If you wish to read CCD data from other institutions the image must be written in FITS format. Otherwise, you must write a program to convert the the data to the format used in VISTA.
2. If you wish to write a package of specialized programs having applications that are not of interest to others, you will need know the formats of VISTA output.

The input and output routines are these:

WRITE	writes images to tape.
READ	reads images from tape.
TAPE	initializes magnetic tape before writing.
TAPEDIR	prints a directory of images on tape.
DISK	reads images from or writes them to the disk.
DISKSP	reads spectra from or writes them to the disk.
SAVE	writes and reads various datafiles. Read the code here to find the format of files for aperture photometry, stellar photometry, brightness profiles, and lists of masked pixels.

VIII. Tailoring VISTA to your own needs

This section reviews some techniques for adjusting VISTA to suit your particular applications.

i) Deleting routines

VISTA is a very large program, containing quite a variety of subroutines for image processing and analysis. It may be that you will never need certain subroutines for your applications. You would then want to reduce the size of VISTA, making it contain only what you want. The procedure for deleting subroutines is:

1. Select the routine you want to delete from VISTA, and find the call to that routine in one of the files composing the command tree.
2. "Comment out" the statements in the command tree pertaining to that command: i.e., put 'C' or '!' at the beginning of those statements, thus turning them into FORTRAN comment lines. It is preferable that you comment out the lines, rather than delete them, because you might want to restore the command someday.
3. Relink VISTA by giving the command

   @[CCDEV.BASE]LINKVISTA

   It is not necessary to alter the linker libraries in any way.

ii) Default directories

VISTA keeps a list of directories and file extensions for various images, spectra, or data files kept on the disk. It automatically searches in these directories unless overridden by the user.

The directory list is created by the subroutine INITIAL, which is executed when VISTA is first run. It reads a set of DCL logical names which specify the directories. If these logical names are defined, VISTA takes them as the names of the default directories. If they are not defined, the program uses the directory structure used at Santa Cruz. File extensions cannot be defined in this way, but may be changed while VISTA is running with the command SETDIR.

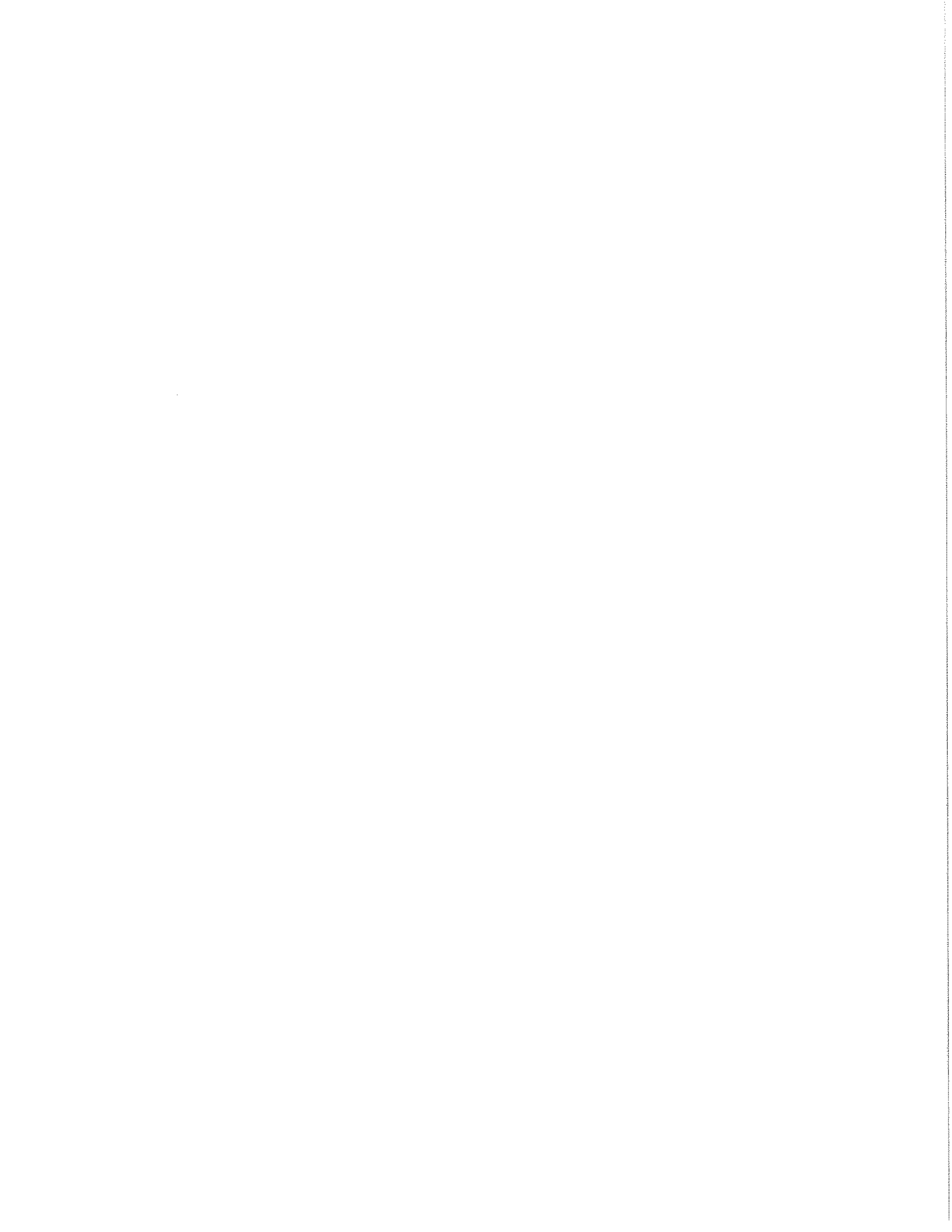
The logical names which INITIAL tries to translate are:

V$CCDIR	images
V$PRODIR	procedures
V$SPECDIR	spectra
V$FLUXDIR	flux calibrations
V$LAMBDIR	wavlength calibrations
V$COLORDIR	color files for TV
V$DATADIR	data files
V$HELPDIR	the help file

Use the DCL command DEFINE to define these logical names. As an example, to re-define the image directory to [MYACCOUNT.MYDIR], type

DEFINE V$CCDIR [MYACCOUNT.MYDIR]

The names of the default directories are stored in common blocks defined in CUSTOMIZE.INC. In addition, VISTA tries to translate V$STARTUP, which defines a file containing a procedure to be run automatically as the program begins. Also the symbols V$LONGITUDE and V$LATITUDE may be used to set the location of the observatory at which images were taken. The longitude and latitude are expressed in decimal degrees.



***** APPENDIX 1 *****

This is a segment of the VISTA command tree, showing the various types of subroutine calls.

```
IF (COM .EQ. ' ') THEN                                                  ! Null
        CONTINUE

ELSE IF (COM .EQ. 'BOX') THEN                                   ! Define boxes
        CALL BOXSET

ELSE IF (COM .EQ. 'MC' .OR. COM .EQ. 'DC') THEN                 ! Multiply or
        NEEDIM  =1                                              ! divide by
        IF (GO) CALL ARITHCON(%VAL(LOCIM),NROW,NCOL)            ! constant

ELSE IF (COM .EQ. 'AI' .OR. COM .EQ. 'SI') THEN                 ! Add or
        NEEDIM  =2                                              ! subtract two
        IF (GO) CALL ARITH2IM(%VAL(LOCIM),NROW,NCOL,            ! images
1                                %VAL(LOCB),JROW,JCOL)

ELSE IF (COM .EQ. 'PSF') THEN                                   ! Point spread fc
        NEEDIM  =-1
        IF (GO) CALL PSF(%VAL(LOCTV),NRTV,NCTV)

ELSE IF (COM .EQ. 'FITSTAR') THEN                               ! Stellar
        NEEDIM = -2                                             ! magnitudes
        IF (GO) CALL FITSTAR(%VAL(LOCTV),NRTV,NCTV,
1                            %VAL(LOCIM),NROW,NCOL)

INCLUDE 'OTHERCOMS.PGM'                                 ! Other commands.

ELSE IF (COM .EQ. 'MASH') THEN                          ! Extract spectrum
        NEEDIM  =1
        IF (GO) CALL MASH(%VAL(LOCIM),NROW,NCOL)

ELSE IF (COM .EQ. 'AS' .OR. COM .EQ. 'SS') THEN                 ! Add or
        NEEDSP  =2                                              ! subtract two
        IF (GO) CALL ARITH2SP(%VAL(LOCS),NSCOL,                 ! spectra
1                               %VAL(LOCSB),JSCOL)

END IF
```

***** APPENDIX 2 *****

This is the file VISTALINK.INC, which contains parameters passed from the top program of VISTA to the subroutines.

```
C       ***     VISTA KEYWORDS, PARAMETERS, AND CONTROL         ***

C       This include file controls communication between VISTA and
C       its subroutines.

C       Parameters:     NCON            The maximum length of the VISTA
C                                       integer, constant, and keyword buffers.
C                       MAXPRO          The maximum number of lines in the
C                                       VISTA procedure buffer.

        PARAMETER (NCON=15, MAXPRO=100)

C       These variables pass constants and keywords to the subroutines.

C       Variable:       CONST   Floating constants buffer
C                       IBUF    Integer constants buffer
C                       WORD    Keywords and alphanumeric buffer
C                       COM     Current VISTA command
C                       COMMAND Full input command line

        DIMENSION CONST(NCON), IBUF(NCON)
        CHARACTER*80 WORD(NCON), COM, COMMAND
        COMMON /VISCON/ CONST, IBUF
        COMMON /VISALPH/ COM, COMMAND, WORD

C       These logical variables control subroutine program flow.

C       Variable:       GO      Set by VISTA to start the subroutine
C                       CHECK   Set .FALSE. in subroutines to indicate
C                               an error in input before VISTA sets GO.
C                       XERR    Set .TRUE. in subroutines when an error
C                               occurs during execution.  If the error
C                               is severe enough, the routine should
C                               imediately return to VISTA.
C                       NOGO    Set .TRUE. when Ctrl-C is entered from
C                               the keyboard.  Subroutines can check
C                               this at any desired point, and return
C                               if set.

        LOGICAL GO, CHECK, XERR, NOGO
        COMMON /CON/ GO, CHECK, XERR, NOGO

C       Author: Tod R. Lauer            11/5/82
```

*** APPENDIX 3 ***

This is the file IMAGELINK.INC, which holds parameters that describe images in VISTA.

```
C       ***     IMAGE PARAMETERS, ADDRESSES, AND CONTROLS        ***

C       This include file contains common blocks holding image
C       parameters, labels, and virtual addresses.

C       The MAXIM parameter controls the maximum number of images that
C       can be handled by VISTA.

        PARAMETER (MAXIM=10)

C       The MAXR and MAXC parameters give the maximum number of columns
C       and rows in a CCD image.

        PARAMETER (MAXR=1024, MAXC=1024)

C       Each image has a list of parameters specifying such things
C       as its size, conditions of observation, reduction history,
C       identifications, and other information deemed important.
C       The list for each image is held in the array HEADBUF.
C       The HEADBUF for each image is divided into a set of 72 'cards',
C       each of which is 80 columns long.  Each parameter is located
C       on a seperate card, identified by a keyword at the start of the
C       card.

        CHARACTER*5760 HEADBUF(MAXIM)
        COMMON /IMGOBJ/ HEADBUF

C       VISTA images are arrays of 2-D floating-point numbers.  The
C       size of the array in each dimension, origin in each dimension,
C       and compression factor in each dimension, are extracted from
C       HEADBUF and held in the integer array ICOORD for convenience.
C       The locations of the various parameters in ICOORD are defined
C       by the following parameters.

        PARAMETER       (N_ROW = 1)
        PARAMETER       (N_COL = 2)
        PARAMETER       (I_SR  = 3)
        PARAMETER       (I_SC  = 4)
        PARAMETER       (I_CMPR= 5)
        PARAMETER       (I_CMPC= 6)

        DIMENSION ICOORD(6,MAXIM)
        COMMON /IMGH/ ICOORD

C       Each image connected with VISTA is assigned an integer image number
C       which is greater than 0 and less than or equal to MAXIM.  An array
C       of logical variables and addresses define the current state of each
C       image number.

C       Variable:       BUFF(IM)        Set .TRUE. if the image number 'IM'
C                                       is connected.
```

```
C                           LOC(IM)           Contains the virtual address of image
C                                             number 'IM'.
C                           NBYTE(IM)         Contains the number of bytes occupied
C                                             by the image.
C                           BUFOLD            Set .TRUE. if a block of virtual
C                                             memory is to be returned.
C                           LOCOLD            The address of the returned memory.
C                           NBOLD             The ammount of returned memory.
C                           IMCREA            New image buffer number

        LOGICAL BUFF(MAXIM), BUFOLD
        INTEGER LOC(MAXIM), NBYTE(MAXIM)
        COMMON /IMG/ BUFF, LOC, NBYTE, BUFOLD, LOCOLD, NBOLD, IMCREA

C       Parameters can be passed to subroutines by use of the IMGPIPE common.
C       The common holds the variables needed to specify the condition and
C       origin of the image data array for a maximum of two images, numbered
C       'IM' and 'JM'.

C       Variable:           ISR               Image starting row >=0
C                           ISC               Image starting column >=0
C                           IRBX              Row compression factor >=1
C                           IRCX              Column compression factor >=1

        COMMON /IMGPIPE/ ISR, ISC, IRBX, ICBX, IM, JSR, JSC, JRBX, JCBX, JM

C       Author: Tod R. Lauer     11/23/82
```

***** APPENDIX 4 *****

Here is a sample VISTA routine.

```
        SUBROUTINE ARITH2IM(A,NROWA,NCOLA,B,NROWB,NCOLB)

C       Routine to add, subtract, multiply and divide two images

C       This routine handles all double image arithmetic.
C       Images are compared to find their region of overlap, based
C       on their origins, offsets, and boxes.  No operation is carried out
C       on portions of the image that do not overlap.  The signs
C       of the offsets are such that rows or columns in B match with
C       the same row and column numbers in A PLUS the offsets.

C       Commands:         AI      Add image B to image A
C                         SI      Subtract image B from image A
C                         MI      Multiply image A by image B
C                         DI      Divide image A by image B

C       Keywords:         DR=n    Offset image B by 'n' rows
C                         DC=n    Offset image B by 'n' columns
C                         BOX=n   Only use portion of image B within box
C                         DARK    Subtraction is scaled by exposure
C                         FLAT    Division is rescaled by image B mean

C       Author: Tod R. Lauer    10/17/82

        INCLUDE 'VISTALINK.INC'                 ! Communication with VISTA
        INCLUDE 'IMAGELINK.INC'                 ! Image parameters
        DIMENSION A(NCOLA,NROWA), B(NCOLB,NROWB)
        CHARACTER*8 PARM
        INTEGER ROW, COL, DX, DY, SR, ER, SC, EC, BN
        LOGICAL DARK, FLAT

C       Check command string

        IF (.NOT. GO) RETURN

C       Select parameters

        DX      =0
        DY      =0
        BN      =0
        F       =1.0
        DARK    =.FALSE.
        FLAT    =.FALSE.
        DO I=1,NCON
                IF (WORD(I)(1:3) .EQ. 'DC=') THEN       ! Column offset
                        CALL ASSIGN(WORD(I),F,PARM)
                        IF (XERR) RETURN
                        DX      =JNINT(F)
                END IF

                IF (WORD(I)(1:3) .EQ. 'DR=') THEN       ! Row offset
                        CALL ASSIGN(WORD(I),F,PARM)
                        IF (XERR) RETURN
                        DY      =JNINT(F)
```

```
            END IF

            IF (WORD(I)(1:4) .EQ. 'BOX=') THEN          ! Subsection specified
                    CALL ASSIGN(WORD(I),F,PARM)
                    IF (XERR) RETURN
                    BN       =JNINT(F)
            END IF

            IF (WORD(I) .EQ. 'FLAT') FLAT    =.TRUE.

            IF (WORD(I) .EQ. 'DARK') DARK    =.TRUE.

        END DO

C       Compare the dimensions, origins, and offsets of the two
C       images, to find their overlap.  The arithmetic will only
C       be performed on that section.

        IF (IRBX .NE. JRBX .OR. ICBX .NE. JCBX) THEN
                TYPE *,'The images have differing compression factors...'
                XERR     =.TRUE.
                RETURN
        END IF

C       Get box parameters

        IF (BN .EQ. 0) THEN
                JBSR     =JSR
                JBSC     =JSC
                NBROWB   =NROWB
                NBCOLB   =NCOLB
        ELSE
                CALL GETBOX(BN,ICOORD(1,JM),SR,SC,ER,EC,NBROWB,NBCOLB)
                IF (XERR) RETURN
                JBSR     =SR+JSR-1
                JBSC     =SC+JSC-1
        END IF

        SR      =MAX0(ISR-DY,JBSR)+1-JBSR                  ! Starting row
        ER      =MIN0(ISR+NROWA-DY,JBSR+NBROWB)-JBSR       ! Ending row
        SC      =MAX0(ISC-DX,JBSC)+1-JBSC                  ! Starting column
        EC      =MIN0(ISC+NCOLA-DX,JBSC+NBCOLB)-JBSC       ! Ending column
        DY      =DY+JBSR-ISR
        DX      =DX+JBSC-ISC
        IF (ER .LT. SR .OR. EC .LT. SC) THEN
                TYPE *,'The images do not overlap...'
                XERR     =.TRUE.
                RETURN
        END IF

        CALL SUBVAR('M',IM,AV1,.FALSE.)          ! Get image means for later
        CALL SUBVAR('M',JM,AV2,.FALSE.)          ! updating

C       Branch to specified operation

        IF (COM .EQ. 'AI') THEN

C       Add the images

        DO ROW=SR, ER
                DO COL=SC, EC
```

```
                        A(COL+DX,ROW+DY)=A(COL+DX,ROW+DY)+B(COL,ROW)
                END DO

        END DO

        AV1     =AV1+AV2
        CALL SUBVAR('M',IM,AV1,.TRUE.)

        ELSE IF (COM .EQ. 'SI') THEN

C       Perform normal or dark subtraction

        IF (DARK) THEN                          ! Scale subtraction by exposure
                T1      =FHEAD('EXPOSURE',HEADBUF(IM))
                T2      =FHEAD('EXPOSURE',HEADBUF(JM))
                F       =T1/T2
                DO ROW=SR, ER
                        DO COL=SC, EC
                        A(COL+DX,ROW+DY)=A(COL+DX,ROW+DY)-F*B(COL,ROW)

                        END DO
                END DO

                AV1     =AV1-F*AV2
                CALL SUBVAR('M',IM,AV1,.TRUE.)

        ELSE                                    ! Normal subtraction
                DO ROW=SR, ER
                        DO COL=SC, EC
                        A(COL+DX,ROW+DY)=A(COL+DX,ROW+DY)-B(COL,ROW)

                        END DO
                END DO

                AV1     =AV1-AV2
                CALL SUBVAR('M',IM,AV1,.TRUE.)

        END IF

        ELSE IF (COM .EQ. 'MI') THEN            ! Multiply images
        DO ROW=SR, ER
                DO COL=SC, EC
                        A(COL+DX,ROW+DY)=A(COL+DX,ROW+DY)*B(COL,ROW)
                END DO

        END DO

        IF (AV2 .NE. 0.0) AV1=AV1*AV2
        CALL SUBVAR('M',IM,AV1,.TRUE.)

        ELSE IF (COM .EQ. 'DI') THEN

C       Check for zeros and then perform division.  Pixels where divide
C       by zero is attempted are zeroed out themselves.

        IF (FLAT .AND. AV2 .NE. 0.0) THEN       ! Scale factor is image 2 mean
                F       =AV2
        ELSE IF (FLAT) THEN                     ! Calculate image 2 mean
                SUM     =0.0
                DO ROW=SR, ER
                        DO COL=SC, EC
```

```
                                SUM       =SUM+B(COL,ROW)
                        END DO
                END DO
                F               =SUM/FLOATJ((ER-SR+1)*(EC-SC+1))
                CALL SUBVAR('M',JM,F,.TRUE.)
                IF (F .EQ. 0.0) F=1.0
        ELSE
                F               =1.0
        END IF

        DO ROW=SR, ER
                DO COL=SC, EC
                IF (B(COL,ROW) .EQ. 0.0) THEN
                        A(COL+DX,ROW+DY)=0.0
                ELSE
                        A(COL+DX,ROW+DY)=F*A(COL+DX,ROW+DY)/B(COL,ROW)
                END IF
                END DO

        END DO

        IF (AV2 .NE. 0.0) AV1=F*AV1/AV2
        CALL SUBVAR('M',IM,AV1,.TRUE.)
        END IF

        RETURN
        END
```

**** APPENDIX 5 ****

Shown here is a segment from the VISTA helpfile.

```
:ARITHMETIC ON IMAGES AND SPECTRA
.AI ADD TWO IMAGES
.SI SUBTRACT TWO IMAGES
.DI DIVIDE TWO IMAGES
.MI MULTIPLY TWO IMAGES
/Addition//Two images
/Subtraction//Two images
/Multiplication//Two images
/Division//Two images
/Image//Add another image
/Image//Subtract another image
/Image//Multiply by another image
/Image//Divide by another image

        The commands that perform arithmetic between two images are:

AI dest source [BOX=n] [DR=n] [DC=n]        (dest=dest+source)
SI dest source [BOX=n] [DR=n] [DC=n] [DARK] (dest=dest-[Ts/Td]*source)
DI dest source [BOX=n] [DR=n] [DC=n] [FLAT] (dest=dest/source[*source mean])
MI dest source [BOX=n] [DR=n] [DC=n]        (dest=dest*source)

        where:

        dest            (integer or $ construct) is the buffer
                         holding the one image and also specifying
                         the buffer where the result will be stored,

        source          (integer or $ construct) is the other
                         image used in the arithmetic,

        BOX=n           uses only that portion of the source image that is
                        in box 'n',

        DR, DC          shifts the 'source' image before doing the arithmetic.

$PAUSE
{AI dest source [BOX=n] [DR=n] [DC=n]        (dest=dest+source)
{SI dest source [BOX=n] [DR=n] [DC=n] [DARK] (dest=dest-[Ts/Td]*source)
{DI dest source [BOX=n] [DR=n] [DC=n] [FLAT] (dest=dest/source[*source mean])
{MI dest source [BOX=n] [DR=n] [DC=n]        (dest=dest*source)
{
        Note that the result of the arithmetic operation is stored
in the first location mentioned on the command line!  The syntax of
these commands is

        OPERATE ON (image 1) WITH (image 2)

        The operations are done on a pixel-by-pixel basis so that
pixel (I,J) of the 'dest' image is combined with pixel (I,J) of the
'source' image. 'DR' and 'DC' can be used to specify an optional
offset in number of rows or columns of the source image when it
operates on the destination. 'DR' and 'DC' can be negative as well as
positive, but will be rounded to the nearest integer.  The result is
```

that row 'I' and column 'J' of the source operates on row 'I+DR' and column 'J+DC' of the destination. IMPORTANT: If the images do not overlap exactly, only those pixels in the destination image that are also in the source image are effected by the operation. The other pixels are not changed!

$PAUSE

```
{AI dest source [BOX=n] [DR=n] [DC=n]          (dest=dest+source)
{SI dest source [BOX=n] [DR=n] [DC=n] [DARK]   (dest=dest-[Ts/Td]*source)
{DI dest source [BOX=n] [DR=n] [DC=n] [FLAT]   (dest=dest/source[*source mean])
{MI dest source [BOX=n] [DR=n] [DC=n]          (dest=dest*source)
{
```

If the optional keyword FLAT is included in the DI command, the resulting image is multiplied by the mean of the image in the 'source' buffer. The mean is NOT computed by the DI command, but must be computed with the MN command ahead of time. If the optional keyword DARK is included on SI command line, then the source image is scaled by its exposure time relative to the destination image before the subtraction. These two operations preserve the mean of the destination image.

Examples:

AI 2 1	adds image 2 to image 1 and stores the result in image 1.

$PAUSE

DI 2 3 FLAT	divides image 2 by image 3, then scales the result by the mean of image 3. This preserves the mean of image 1.
AI 4 7 DR=4 DC=-10	add image 4 to image 7, bur first shift image 7 by 4 rows and by -10 columns. The pixels in image 4 that are also in image 7 will contain the sum. The pixels in image 4 that are NOT in image 7 will not be changed.
AI $IM1 $IM2	add image IM1 to IM2, where IM1 and IM2 are variables. (This is helpful in procedures.)