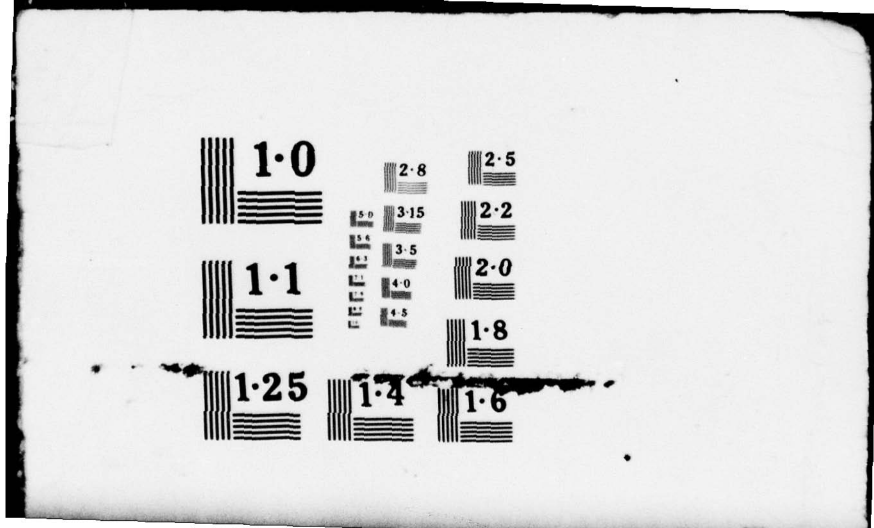
AD-A072 405 NAVAL RESEARCH LAB WASHINGTON DC F/G 9/2
A USERS GUIDE FOR THE ADVANCED SCIENTIFIC COMPUTER (ASC).(U)
JUN 79 W W JONES
UNCLASSIFIED NRL-MR-3979 NL

1 OF 1
AD
A072405

END
DATE
FILMED
9 - 79
DDC


1·0
2·8
2·5
3·15
2·2
3·5
2·0
4·0
1·1
4·5
1·8
1·25
1·4
1·6

AD A072405

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NRL Memorandum Report 3979	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) A USERS GUIDE FOR THE (A)DVANCED (S)CIENTIFIC (C)OMPUTER (ASC).		5. TYPE OF REPORT & PERIOD COVERED (9) Interim rept.
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) Walter W. Jones		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Research Laboratory Washington, D. C.		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NRL J. O. 62H02-51 NRL Task No. SSP0/1781319.3083 ONR Task No. RR04-09-41
11. CONTROLLING OFFICE NAME AND ADDRESS (11) 26 Jun 79		12. REPORT DATE June 26, 1979
		13. NUMBER OF PAGES 43
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) (12) 44 p.		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited. (16) RR0409

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

(17) RR040941

18. SUPPLEMENTARY NOTES

This project was sponsored by Office of Naval Research, Project No. RR04-09-41.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computers
Advanced Scientific Computer
Debugging

(14) NRL-MR-3979

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A great deal of effort goes into starting on a new computer system. This is especially true on sophisticated system designed to accomplish intricate tasks. In order to minimize this effort and thus to maximize people usefulness this guide has been prepared for the NRL ASC computer.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

251 950

CONTENTS

I. INTRODUCTION 1

II. LIBRARIES 2

(A) PROGRAM 3

(B) MACRO 4

III. CATALOG STRUCTURE 5

IV. JOB SPECIFICATION LANGUAGE 7

V. JOB STRUCTURES – WHO DOES WHAT 8

VI. RUNNING A JOB – BASIC 10

VII. RUNNING A JOB – ADVANCED 15

VIII. ERRORS AND DEBUGGING 20

IX. ACKNOWLEDGMENT 25

APPENDIX I – EXPLANATION OF MACROS 25

APPENDIX II – VECTOR PARAMETER FILES 30

Accession For
NTIS GRA&I
DDC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist
Avail and/or special
A

A USERS GUIDE FOR THE (A)DVANCED (S)CIENTIFIC (C)OMPUTER

I. Introduction

The purpose of this user's guide is to acquaint both the beginning user and the advanced user with information that is not readily available from other sources concerning operations of the ASC computer. It is not intended to replace any of the Texas Instruments Software manuals nor to represent itself as being a complete guide. It will, however, provide pointers to places where more information can be obtained.

As can be seen from the table of contents, the subject separates into several distinct catagories. The first try at using this computer might well be to use Chapter VI and submit a job. Once the initial shock of actually having output returned is over, one can proceed with the problem of utilizing the facilities that are available.

The ASC computer (manufactured by Texas Instruments) is a general purpose vector computer with the additional feature that instructions are processed in a pipe-line mode. This simply means that the box that does the actual work is looking at more than one instruction. In fact, it can be working on as many as four instructions simultaneously, and, unless a branch causes some instructions to be discarded, produces one result each machine cycle. Since the NRL version has two arithmetic units, the computer can be processing up to eight instructions at one time. Such an architecture enables the hardware to make the most efficient use of the cables that connect the various parts of the computer. If processing were not done in this manner, then the computer would stand idle part of the time, namely that part of the time when it is having to fetch operands after it has interpreted an instruction. As it can operate, however, by the time the arithmetic unit is ready to execute a particular instruction, the data is already available.

The vector nature of the computer is simply the ability of the arithmetic unit to intrepret only one instruction and perform this operation on a large set of data. This makes for more efficient utilization in that fewer machines cycles need to be devoted to interpreting instructions, and more can be spent doing computations.

Manuscript submitted April 26, 1979.

II. Libraries

There are primarily two types of libraries. They are the program library and the macro library. Program libraries can be source, object or load module (run time) libraries. The most common of these are source and object module libraries. Load module libraries will not be dealt with extensively in this guide.

The following two sections (II.A and II.B) deal with and explain these two types of libraries in some detail as well as giving a brief description of macro libraries. The actual structure that is used by the operating system that handles the libraries will not be discussed since it is not important in using these facilities. What is important, however, is to understand what the libraries can contain, so that maximum use can be made of them.

Disk and tape files, in general, come in three flavors, namely sequential (PS), partitioned (PDS), and direct secondary (core image) files (DS). Sequential (physical sequential or PS) files are equivalent to having the program (set of cards) in one long tray. Partitioned (partitioned direct secondary or PDS) files are similar to having a library of books where one must refer to the card catalog first in order to find the location of the book. This latter structure allows one to work on individual programs (books) without having to keep track of all other programs. An analogy might be the individual books in an encyclopedia; it is much easier to change one book and reprint it if an error is found than would be the case if the whole lot were bound together whence the entire encyclopedia would have to be reprinted each time an error were found. Experience has shown that errors often arise when these file types are confused. Direct secondary (DS) files will not be dealt with in this guide, since typical users seldom have need to manipulate them directly.

Physical sequential files contain only one module, whereas partitioned files, by their very nature, contain one or more modules with a concomitant directory. Discussion of sequential files can be found in the "Sequential I/O User's Guide" (#930055) and information on PDS files can be obtained from the manual "Partitioned Direct Secondary Files" (#931487).

In general, program files should be maintained in the PDS format to facilitate updating

programs. In some specialized cases, however, physical sequential files are used. When program files are organized in the PS format, they are referred to simply as files or modules. When they are organized according to the PDS format, then the program element is called a module and is a member of the library.

II.A. Program Libraries

Program libraries contain three types of programs, that is, programs in three forms. These are source, object and load module libraries. Source and object files can be organized in either of the first two forms of disk files (PS or PDS), whereas load modules must be organized in the direct secondary or DS format.

Source modules are card image format and there are two principle editors to deal with these: CIFER (#930032) and SMS (#931485). CIFER [Card Image File EditoR] is the more commonly used of the two editors and is quite simple in its use. But it leaves the entire card image on the disk file and thus the space used for blanks is wasted. SMS on the other hand compacts files and in addition allows for systematic updating, but is much more difficult to learn initially becaue of the lack of examples. A few examples of the use of CIFER and SMS will be given in the chapter on examples (VI and VII). Source modules, in general are just like card decks. Any information that can be contained in a card deck can be contained in a source program module, including Job Specification Language cards. The principle use of source modules, however, is to contain the source for a program. A source module can be changed by use of an editor and recompiled or submitted directly for use as a job, depending on the mode in which one is operating.

Object modules contain only the output from language translators. The language translators translate source code into machine language code. The specific form of an object module is not important, but it can be found in the linkage editor manual (#930057). Object modules can be saved when the source is not to be changed from one run to the next and the user does not wish to spend money recompiling the program. These modules (elements of the library) are used only by the linkage editor to create a program which can be excuted directly on the

ASC. The module is stored in card image format, and is directly comparable (and compatible) with a similar card deck. Input to the linkage editor can be from disk, tape or cards. These modules are not excutable directly, but rather, they must be linked, that is, all external references must be found prior to execution. An example of an external reference would be reference to another subroutine or to the Fortran Input Output routines.

II.B. Macro Libraries

Macro libraries contain macros. Macros are simply sets of JSL (see Section IV) which are used repeatedly, and with few, if any, changes from execution to execution. Examples of macros are MFTN, MLNK and MFXQT. If one examines the expansion (in the JSL summary of a job) of a macro, one will see that a given line (or card) can produce many lines of JSL. The detailed, expanded, JSL includes Operating System directives which are environment dependent and usually do not concern the user directly (as always, there are exceptions to this rule). For example, Load Time Parameters for the NX compiler (FORTRAN language translator) are specified in this manner. It is more flexible to have such a capability and implement the *environmental specifications* (what machine, operating system, etc.), by macros rather than building the specifications into each program.

Macro libraries are similar to the PDS program libraries in that they too have a directory to keep track of the various modules. The most important difference is in accessing these libraries. For macros, one must use the

/ MACASG ....

rather than the usual

/ ASG............

command (read the next two sections on JSL to understand the difference). For details on the system macros, that is, the defaults and details of what they do, see "JSL System Macros" manual (#930034), and the R.C.C. computer notes #105 and #137.

Macros are accessed by the JSL translator (see Section IV & V) in a fashion similar to that used by language editors (CIFER, SMS, etc.) to find programs in PDS libraries. When the translator encounters a JSL name, the macro name list is searched to determine if the name refers to a macro. The list is set up (or modified) when a MACASG verb is encountered. This list of macro names can be changed (this feature is useful mostly for interactive users) by releasing the macro file together with the file SYS.JSLM.

Examples of macros and their use will be given in the section on examples (VI and VII) and a more detailed explanation is given in Appendix I.

III. Catalog Structure - Where Are Your Programs

The catalog system is simply a means for the computer to store your programs and remember where they are located. The operating system is designed so that the use of the catalog is device independent. It works quite well except for one minor problem in using tape files in which case disk space must be allocated for the tape staging - see the section on examples. The difference between disk and tape files is mostly locally imposed (different charge rates) except that to an interactive user, a tape mount is noticeably slower than a disk access. With the catalog structure as simple as it is, there is little need to use large card decks to run jobs. Programs are more properly stored on disk or tape, and accessed via the catalog structure.

The terminology of the catalog structure format is straight-forward. A root node is the first node. Users cannot change this. Examples of root nodes are S$SYSCAT, USERCAT and AFFIL. One creates nodes, and a path connects the root node to the node of interest. The names themselves are called edgenames. Generally users start working with the forth node. As an example,

USERCAT/D77/L50/MACROS

represents a path with nodes at USERCAT, D77, L50 and MACROS. D77 is a son of the root node USERCAT and is the father of L50, etc.

A node by itself contains nothing. One must create VERSIONS at a node. This is the substance of the catalog structure. Program files, etc., are versions at a node. Versions can be on disk [head per track (HPT) or positioning arm disk (PAD)] or tape. These are the only media currently supported by the NRL ASC operating system although other mass storage system do exist. If several versions exist, then some can be on tape and some on disk. Note that a node which has no versions associated with it contains a null set.

Various attributes are associated with nodes and versions. The attributes associated with a node allow one to control access to the information which is stored there. For example, the user can restrict access to information at the node, and/or reference beyond the node to the sons. The "(J)ob (S)pecification (L)anguage" manual (#930038) deals with this subject, and various interactions in great detail. For the most part, users will not need these facilities, but they are available if the information is sensitive in some way, or simply if the user does not want random people accessing the files. The R.C.C. note #160 also contains some information on access control. Versions also have attributes, but these describe the format in which the information is stored. A particular version, for example, can be in physical sequential format with logical records of 137 bytes and 30 records per block. This type of information would be contained at the version level.

In order to store information, a node must first be created and then versions can be stored at the node. Only one level of adding or deleting can take place with a single line of JSL. Up to sixty-four versions can exist at a node. Such a feature is most useful for maintaining various edited versions of a program. Each successive version can be an updated version of the old program or program file, for example.

An alternative to using the operating system to keep track of one's programs is for the user to do it himself. This can only be done on tape, however. Files which are created (any of those disk file organizations qualify and can be done this way) can be put on tape by the user, and retrieved in a similar fashion. Implementation is via the FIT and FOT commands (see Section IV). The user must then keep track of the files in a manner similar to that used by the operating system for the catalog structure.

IV. Job Specification Language

The Job Specification Language (JSL) is the means of telling the operating system what is to be done with a job. In a sense, it is a programming language, albeit a fairly primative one. The statements specify what is to be done at each step in a job. Actually, the job stream consists of JSL verbs and macros. The JSL translator decodes these statements into something the operating system can understand, which are called IJSL FRAMES. For all practical purposes, JSL verbs and macros serve the same function, and thus macros can, for the most part be treated as if they were JSL verbs. JSL verbs tell the operating system what to do. As an example, the FD, or file definition verb, specifies some information to be passed through to the operating system concerning the characteristics of a file, such as the type of organization of the file. Referring back to an earlier section, the appropriate keyword to define the type of file on the FD statement would be FORG=PS, PD or DS, which tells the operating system that the file is organized according to the physical sequential, partitioned direct secondary, or the direct secondary organization.

The JSL verbs which work on the NRL ASC (#7) are

ASG, ASGP, CAT, CATBLD, CATN, CATP, CATV, CHG, DEL,
DELV, FD, FIPT, FIPTP, FIT, FITP, FOPT, FOPTP, FOT, FOTP,
JOB, LIMIT, MACASG, MACBLD, MFR, MFRE, PJSL, RPLV, START,
STOP, XQT, RENAME and REL

and the system macros are

ASM, ASMC, ASML, ASMLX, ASMP, CATLST, CIFER, CJSL, DOCUMENT,
DSDUMP, DUMPER, EOJ, FILECNVRT, FILECOMP, FILECOPY,
FILEPTCH, FILLST, FOSYS, FTN, FTNL, FTNLX, FXQT, LMPATCHER,
LNK, LNKX, NEWFILE, NEWLIB, NEWPRINT, NEWPUNCH, OPTPRT$$, PDNLYZ,
PDSCOL, PDSDIR, PDSLST, PDSQSH, PDSUPD, PLAP, SMS, SMSASM,
SMSASMC, SMSASMP, SMSFTN, SORT, and TPNLYZ.

System macros and JSL verbs which are not available (or do not currently work) on ASC (#7) are

BATINT, CNT/CNTE, SETUP, CFSTEP, GET, GETCAT,
GETMEM, MERGE, PRNT, PUNCH, SAVE, SAVECAT, SAVEMEM, and SPLIT

Details on the defaults for these verbs and macros can be found in the "Job Specification Language" manual (#930038) and the "JSL System Macros" manual (#930034). The R.C.C. computer note #105 also has the locally imposed defaults listed. This too is a useful reference manual.

V. Job Structure — Who Does What

A JOB is structured by dividing it into parts which the operating system considers similar, and therefore can handle, in the sense of not biting off more than it can chew. This may not be identical to a programmers idea of similarity, but in order to understand the various phases of a job, the "OS" point of view must be considered. A schematic of this breakdown is shown in Fig. (1).

A JOB is defined by a JOB and an EOJ pair. In between there are phases which are defined by LIMIT cards (or the JOB/EOJ pair if the limit defaults are used) as a block. The usual block consists of a compile followed by a link and then an execute step. This is not fixed, however, and several compiles, links and execute steps can be combined within one job block, once again, defined by LIMIT cards. Alternatively, each step can be in its own block. It should be pointed out that the resources requested affect both the turnaround of a job, and the amount it costs to run that job. Roughly, the more resources requested, the greater the cost, and the longer the wait.

The various phases have to run through both a scheduler (mix table) and a real-time scheduler which actually fits the requirement of the job into the system and runs the job. In a simplified manner, the order of processing a phase of a job is as follows:

(1) Accept the job from an input device
(2) Scan the blocks for resources required and grab them when they are available (all at the same time) — does not apply to memory requests
(3) Process the EXECUTE steps of a job
(4) Terminate a job by spooling appropriate files to the output devices.

For the interactive jobs, a block is the set of JSL which is appended, [AP, FILE] at the terminal.

The job is accepted by an input device, such as a card reader, and is put into the initiation queue, called the JIQ. As soon as the operating system can begin to search for the requested resources, then the job is moved to the preprocessing list, or PPL. The determination to move a job from the JIQ to the PPL is made by the scheduler on the basis of information given to it by the job, rather than on the actual resources which are to be used. If the job has lied at this point, there is a good chance that the PPL scheduler will cancel the step since it won't be able to find the necessary resources. This applies both to batch and interactive job. At the PPL, the system begins to grab the resources necessary to run the job. It is at this point that tapes are fetched from the library. A job will not leave this queue until all resource requests (except central memory) are satisfied. This is to prevent a job from becoming active in the central processor and then not being able to obtain a tape or disk, which could result in a hung system.

Once the resource requests are satisfied, the job is moved to the active job list, or AJL. It is at this point that the real time scheduler attempts to fit the job into the space which it has available, namely the maximum central memory and the total wall clock time available during the day. When a job is on the AJL, it can be waiting to use some facility, such as central memory, or it can actually be executing. In this latter case, the system will indicate that the job is in STEP. After all steps have completed, the job goes to the JBTL list where it is ready to be spooled to the output devices. As soon as post processing has finished, such as temporary files being released, and output files (FOSYSed files) moved to the output queue, the job is passed to the JTBO queue, or job to be output list. At this point the remote system (RJE) has access to the job and print, punch and plotting are done as necessary. Interactive jobs are handeled in a similar fashion, except they loop from IJL to PPL to AJL and back, and the mini-computer (a TI 980) which interfaces the terminals to the ASC, maintains its own list of active jobs. Figure (1) shows the interaction of the job blocks for both batch and interactive jobs.

The workload of the ASC is divided among it various components. The mainframe itself is divided into central memory, the peripheral processor and the central processor. The central memory is the main storage unit where the operating system, as well as programs which are ex-

ecuting, reside. The central processor, which consists of a memory buffer unit (MBU) and two arithmetic units (pipes) handles the JSL translator as well as user programs. When a program is in STEP in the AJL, it is resident in central memory and the central processor is taking instructions from the central memory in an order determined by the programmer. A schematic diagram of the main components of the ASC is shown in Fig. (2).

Except for the JSL translator, the operating system is run by a unit known as the peripheral processor. Although the operating system instructions are in central memory, the actual work is done by the PP which is a separate component. This is the unit which handles INPUT/OUTPUT requests, transfers data from disk-to-tape, and other non-computation oriented requests. It also does the job scheduling and account processing. This computer runs in parallel with the central processor and the only interaction is through the central memory and some special control registers. Some details of these various units and their interaction can be found in the manuals "The Programers Guide to the Peripheral Processor" (#930040), "The ASC Operating System" (#0545) "Description of the ASC" (#934662-934666) and "Description of the ASC Hardware" (#929970).

VI. Running a Job—Basic

The main goal in using a computer is being able to submit a program and to obtain results with a minimum of fuss and bother. Some examples are given in this section which should enable a user to submit a job and have it work (barring typing errors of course). It is assumed that the user is familiar with the programming language FORTRAN. This is the primary high level language in use on the ASC, and will thus be used for examples. Other languages which are available are ALGOL (algol 60), PASCAL, COBOL, ASC CP and PP assembly language and POPS; soon to be available will be some list processing languages such as SNOBOL or LISP.

Three examples are given below. Much of the intermediate output which the user will see is omitted here as not being relevant to understanding this first pass through the world of the ASC. Much of it is useful, however, in understanding more advanced examples and for optimization and debugging, so take a look at it now, even if a very cursory look.

The first example deals with a self-contained compile, link and execute step.

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1)  /  JOB JOBNAME,ACCOUNT NO.,USERCODE,OPTIONS
2)  /  MACASG MACRX$,USERCAT/D77/L50/MACROS
3)  /  MFTN COMPILER=FX
4)            PROGRAM EXAMPL
5)            CALL COMMON
              WRITE (6,101)
       101    FORMAT ('  FOR THE FIRST TRY, X = ')
              X = 3.0
              CALL SUBA (X)
              STOP 987
              END
6)            SUBROUTINE SUBA (ARG)
              WRITE(6,101) ARG
       101    FORMAT ('+',23X,F6.2)
              RETURN
              END
7)  /  MLNK
8)  /  MFXQT
9)  /  EOJ
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

One line should be printed by this program, namely

FOR THE FIRST TRY, X = 3.00

#1 The job card.
#2 Assigns the special macros discussed in Appendix I.
#3 Invokes the fast compiler (FX) on the program following it.
#4,#5 Specify the name of the program and call the general debugging routine COMMON
#6 The subroutine SUBA, called in the main program, is included here
#7 Invokes the linkage editor to resolve the external references and to set the program up into executable form
#8 Loads and starts the program, which then prints the above line.
#9 Terminates the job—at RJE sites, this must be followed by a card with the 7/9 multipunch in column one.

An extension of this would be if a subroutine were to be fetched from the user's library, as shown in example #2.

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1) / JOB JOBNAME,ACCOUNT NO.,USERCODE, OPTIONS
2) / MACASG MACRX$, USERCAT/D77/L50/MACROS
3) / MFTN COMPILER=NX
4)          PROGRAM E2
            DIMENSION A(10,10)
5)          CALL COMMON
            NX = 10
            NY = 10
            DO 101 J = 1, NY
            DO 101 I = 1, NX
       101  A(I,J) = FLOAT(I+J)
6)          CALL PRNT (' ARRAY - A(I,J) ',A,NX,NY,1,NX,1,NY)
            STOP 123
            END
7) / ASG    PLIB,USERCAT/D77/L50/LIB,USE=SHR
8) / MLNK
9) LIBRARY PLIB
   / MFXQT
   / EOJ
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

This program will print out a 10×10 array of numbers with a heading.

#1-#5	Same as previous example
#6	External subroutine — The linkage editor must fetch this from a library
#7	Tells the operating system to use the access name "PLIB" for the library at the node "USERCAT/D77/L50/LIB"
#8	Same as previous example
#9	Library directive for the linkage editor

The program E2 references the subroutine PRNT which is found at the node USERCAT/D77/L50/LIB. The linkage editor uses the access name PLIB for this library. Any other valid name, such as XYZ12345, could have been used as well. The trick here is that the same access name must be used in the library directive statement (#9) as was used in the assign statement (#7). The third example is that of fetching a source program from a tape file, editing and compiling this program, and after compiling a main program and a subroutine, all with different compilers, and running the program, as in previous examples. In examining the following example, the reader should notice that no *explicit* reference is made to the fact that some of the programs reside in a tape file.

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1)  / JOB       JOBNAME, ACCOUNT NUMBER, USERCODE, OPTIONS
2)  / LIMIT     BAND=40
    / MACASG    MACRX$$,USERCAT/D77/L50/MACROS
3)  / ASG       SO,USERCAT/D77/B40/JONEW1/LIBRARY/SO,USE=SHR
4)  / CIFER
    ..MERGE     SO, PROG1, UPDATE
    .SELECT     PLOTIT
    -1,2
                SUBROUTINE PLOTPS(A,NL,NC)
    -40
    C           THIS IS AN EDITED VERSION OF PLOTIT
                NP = 1
                NZ = 0
                NUL = 56
                NUC = 101
                NSORT = 0
                LEN = 0
                TITLE (1) = SYM (1)
                NPP = 1
                XMAX = -1.E + 70
                YMAX = -1.E + 70
5)  / REL       SO
6)  / START     ACNM=PROG2
                PROGRAM DRIVER
                CALL COMMON
                CALL SUBB
                STOP 777
                END
7)  / STOP
8A) / MFTN      COMPILER=NX, IN=PROG1
8B) / MFTN      COMPILER=FX, IN=PROG2
8C) / MFTN
                SUBROUTINE SUBB
                DIMENSION XY(15,2)
                DO 1 I=1, 15
          1     XY(I,1) = FLOAT(I)
                READ(5,2) (XY(I,2),I=1, 15)
          2     FORMAT(15F4.0)
                CALL PRNT(' RANDOM NUMBERS ',XY,15,2,1,15,1,2)
                CALL PLOTPS(XY,15,2)
                RETURN
                END
9)  / PD LOCALP, USERCAT/D77/L50
10) / ASG PLIB, LOCALP/LIB, USE = SHR
11) / MLNK
    LIBRARY PLIB
12) / MFXQT DATA=WHATEVER
13) / START ACNM=WHATEVER
    0.11.321.117.209.435.100.017.100.173.332.224.446.666.370.45
    / STOP
    / EOJ
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

This example will produce a printout of the data together with a plot of the same data.

#1	Job card
#2	Allocation of 40 bands of HPT (head per track disk) for this job
#3	Assign the source library to the access name SO
#4	Invoke the editor CIFER
#5	Release the source file after we are finished with it
#6-#7	A START/STOP file which contains the main program
#8	Three compile steps with various combination of compilers and input files
#9	A (P)ath (D)efinition to save typing later on
#10	Same as #7 in the previous example
#12	An execute with the input coming from the file WHATEVER
#13	A START/STOP file to define the input data for the file WHATEVER

The extra disk space allocation required by this tape assign (#2) is the problem referred to earlier, namely that tape staging requires the allocation of disk space, since the file does not currently reside on the disk. The user reserves the necessary scratch space with the limit card. Line #5 then releases the file when it is no longer necessary to have it available, so that the disk space can be used later. Note that no explicit reference is made to the fact that this is a tape file, and the only way to know this is to check the job activity list to find the tape number that was used in the mount request for the operator.

This example also gives a short lesson in how to use one of the source file editors, in this case, CIFER. Since the compilers take the programs in PS organization, and a library is in the PDS organization, the editor is used to make the switch via the MERGE command. The particular program that is wanted is SELECTed. The access name for this program becomes PROG1 rather than SO. Also shown are two examples of START/STOP files. The only restriction on a START/STOP file is that another START/STOP file can not be embedded, and JOB/EOJ must be paired. This JSL verb is used to insert PS files into the job stream.

Several types of compiles are shown, with both the NX (the default) and the FX compiler being used. In the first two cases, the programs (PROG1 and PROG2) are defined elsewhere, but for the third compile, the program comes from cards which follows the MFTN card. An example of the PD or path definition is given in line #9. After this JSL verb is used, the name LOCALP can be used wherever the path USERCAT/D77/L50 was used earlier. This particular example does not fully demonstrate the time saving that this command can have, but after a

while, the user will realize the benefit.

These three examples are probably sufficient for a first pass through the world of the ASC computer. Once these examples have been tried, and prior to creating files (examples of which are given in the next section), it would be worthwhile reading the earlier sections on libraries and file structure.

VII. Running a Job—Advanced

It has been stressed repeatedly that the catalog system of the ASC obviates the need for large card decks. As an extension of section (VI) it will be shown how this can be done with the least fuss and bother. CIFER will be the editor used as an example, and it will be used to modify FORTRAN source modules. The technique here, however, is that of manipulating files, and not one of espousing the virtues of one source module editor over another, or one language over another. The examples will start with the process of creating a file, and from that point modifying this file to obtain various results.

The first example begins with the program given in example #3 of the previous section.

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      / JOB JOBNAME, ACCOUNT, USERCODE, OPTIONS
      / LIMIT    BAND = 40
      / MACASG   MACRX, USERCAT/D77/L50/MACROS
1)    / PD       U, USERCAT/D77
2)    / ASG      CIFER, U/L50/CIFER, USE = SHR
      / ASG      SO, U/B40/JONEW1/LIBRARY/SO, USE = SHR
      / CIFER
3)    ..COPY     SO, NEWSO, UPDATE
      .SELECT    PLOTIT
      -1,2       SUBROUTINE PLOTPS (A,NL,NC)
      -40
                 NP = 1
                 NZ = 0
                 NUL = 56
                 NUC = 101
                 NSORT = 0
                 LEN = 0
                 NPP = 1
                 TITLE (1) = SYM (1)
                 XMAX = -1.E + 70
                 YMAX = -1.E + 70
4)    ..SPLIT    *, NEWSO
                 PROGRAM TEST 1
                 CALL COMMON
                 CALL SUBB
                 STOP 777
                 END
      ..PRINT    NEWSO
5)    / PD       MYPATH, USERCAT/Dnn/Bmm/USERC1
6)    / CATN     MYPATH/TEST
7)    / CATN     MYPATH/TEST/SO
8)    / CATV     MYPATH/TEST/SO, ACNM = NEWSO
      / EOJ
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

#1 Path definition to reduce typing in the following two lines.

#2 Assign the new version of the CIFER program because it runs faster and costs less.

#3 A CIFER verb to take a SELECTed source module - in this case PLOTIT - and move it from the library file SO to the library file NEWSO.

#4 A CIFER verb to examine the following cards images (*) and also place them in NEWSO with module names given by the names found in the PROGRAM, SUBROUTINE and FUNCTION cards.

#5 Path definition appropriate to the user.

#6 Define a node called TEST, under the users path.

#7 Define a node called SO, under TEST.

#8 Catalog the library file NEWSO at the node MYPATH/TEST/SO.

- -

This example is designed to illustrate the mechanics of creating a library file of source modules and cataloging it on disk, as a version, at the node

USERCAT/Dnn/Bmm/USERC1/TEST/SO.

The assumption made here that the user has registered an account number at the Work Control Center, and thus has obtained the names which correspond to Dnn, Bmm and USERC1. This notation will be assumed from now on, and the appropriate substitution should be made.

As for the example itself, two facts stand out: first, as noted previously, only one level of catalog structure can be added or deleted with a given line of JSL, and consequently, both lines #6 and #7 are required; secondly, that the user wishes to use the same name for the source modules as is given to the program (line #4). It is a simple matter to use alternative names for the modules by inserting appropriate directive cards just before each program. Such is not done in this example for simplicity: there are already three distinct names to worry about, and adding a forth distinct name to distinguish source modules and program names will not help understanding. The three names are (1) node name (SO), (2) temporary file name (NEWSO), and (3) program and module names (PLOTIT and TEST1). The bottom line is that a source file now exists, with two programs contained in it, in a library format. The next example shows how to add another routine (SUBB) and edit one of the original source modules. These three programs are then compiled, saved as updated source and object files, and the load module is executed.

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     / JOB      .......
     / MACASG        MACRX $$, USERCAT/D77/L50/MACROS
     / PD       U, USERCAT/D77/L50
     / PD       ME, USERCAT/Dnn/Bmm/USERC1
     / ASG      CIFER, U/CIFER, USE = SHR
     / ASG      SO, ME/TEST/SO, USE = SHR
     / CIFER
1)   ..MERGE    SO,SYS.FIN, UPDATE
     .SELECT    TEST1
     .SELECT    PLOTPS
2)   -41,41
                NUL = 26
3)   ..COPY     *, SYS.FIN
                SUBROUTINE SUBB
                DIMENSION XY(15,2)
                DO 1 I = 1, 15
     1          XY (I, 1) = FLOAT (I)
                READ ( 5, 2) (XY (I, 2), I = 1, 15)
     2          FORMAT (15F4.0)
                CALL PRNT ('RANDOM NUMBERS', XY, 15, 2, 1, 15, 1,2,)
                CALL PLOTPS (XY, 15, 2)
                RETURN
                END
4)   / MFTN     COMPILER = FX
5)   / FD       SYS.FIN, POS = NEW
     / FD       SYS.OMOD, POS = NEW
     / CIFER
6)   ..SPLIT    SYS.FIN, NEWSO
     ..SPLIT    SYS.OMOD, NEWOB
7)   / CATN     ME/TEST/OB
8)   / CATV     ME/TEST/SO, ACNM = NEWSO
     / CATV     ME/TEST/OB, ACNM = NEWOB
     / ASG PLIB, U/LIB, USE = SHR
     / MLNK
     LIBRARY PLIB
9)   / MFXQT
     .111.123.9871.110.02.1731.02.101.632.777.851.233.921.651.333
     /          EOJ
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

#1 Reformat the source file from the PDS format to the PS format while copying it to the temporary file SYS.FIN, and doing updates on the modules

#2 Change the number of lines in the output of the routine PLOTPS

#3 Copy the following cards (on card images) into SYS.FIN

#4 Compile all the programs with the FX compiler. Note that the default input for the compiler is SYS.FIN

#5 Positon the file pointer to the beginning of the file for use by CIFER

#6 The CIFER verb to reformat the physical sequential files, SYS.FIN and SYS.OMOD, into the PDS formated files whose names are NEWSO and NEWOB

#7 Create a node called OB under the path ME/TEST

#8 Replace the old contents of SO with the contents of the file NEWSO

#9 Execute the load module which has been created by the MLNK step

This example shows *one* means by which a user can (1) edit a source module, (2) add another source module to the file and (3) run the entire setup. In adddition, the object modules have been saved at the node TEST/OB and will be used in the next example to demonstrate the use of object libraries to save recompiling source modules from one execution to the next.

Another example of manipulating these files is to use the same subroutines with another main program.

```
--------------------------------------------------
    / JOB     ......
    / MACASG MACRX $$, USERCAT/D77/L50/MACROS
    / PD      U,USERCAT/D77/L50
    / PD      MYPATH, USERCAT/Dnn/Bmm/USERC1
    / ASG     CIFER, U/CIFER, USE = SHR
    / ASG     SO, MYPATH/TEST/SO, USE = SHR
#1) / ASG     OB, MYPATH/TEST/OB, USE = SHR
    / CIFER
    ..COPY    *, SYS.FIN
#2            PROGRAM TEST 2
              CALL COMMON
              PRINT 1
    1         FORMAT ('THIS IS TEST #2' )
              CALL SUBB
              PRINT 2
    2         FORMAT ('IF THIS MESSAGE IS PRINTED, THERE IS AN ERROR')
              STOP 69
              END
    ..MERGE   SO, SYS.FIN
#3  .SELECT   PLOTPS
    -41,42
              NUL = 36
              NUC = 51
              STOP 777
    / MFTN    COMPILER = FX
    / ASG     PLIB, U/LIB, USE = SHR
    / MLNK
    LIBRARY OB, PLIB
    / MFXQT
    .123.456.789.123.456.789.456.123.777.888.999.965.386.451.010
    / EOJ
--------------------------------------------------
```

#1 Assign the object file library
#2 Insert a new main program
#3 Modify the plot routine

This example adds a new driver program and modifies the plotting routine to stop inside it rather than returning to the main program. If the update is done incorrectly, a message to that effect will be printed on the output file. The final example shows the means of deleting files once they are no longer needed. The test files will now be deleted, since, once the user has found his way to this point, there is no longer a need for these programs.

```
-----------------------------------------------------
/ JOB    ....
/ PD     ME, USERCAT/Dnn/Bmm/USERC1
/ DEL    ME/TEST/SO
/ DEL    ME/TEST/OB
/ DEL    ME/TEST
/ EOJ
-----------------------------------------------------
```

The only point that needs to be emphasized here is that OB and SO, which are sons of TEST, **must** be deleted prior to deleting TEST.

At this stage, the user should be ready to tackle most JSL problems which arise. The use of more advanced capabilities requires perusal of the various manuals and/or inquiring about the system from an R.C.C. consultant or some other knowledgeable individual.

VIII. Errors and Debugging

Errors fall into three categories: first, there are the simple mistakes such as bad JSL or mispunched data, insufficient disk space or execution time allocation, etc.; next are the normal errors which this section will address in some detail. These first two types of errors comprise about 99% of all problems and includes such idiocy as not initializing variables. Finally, there are the difficult errors which require postmortem dumps. This latter type of error usually accounts for less than 0.1% of the errors and resorting to such drastic debugging techniques should be avoided unless all else fails. It is possible to save postmortem dumps on disk or tape and such a procedure is a useful technique for expensive jobs or those that require a long turn around.

The simple errors are usually a result of overlooking some detail. This type of error can be ferreted out by looking through the activity file and the output from each of the steps. Usu-

ally problems such as forgetting to allow enough disk space, step time or even not having the correct program available can be found here. Most of these errors can be solved rather easily although occasionally one encounters an error message or diagnostic which is quite obscure.

The first step is to examine the job's activity file (SYS,JATF). Termination codes (referred to as TERM CODES or TC) greater than zero are likely candidates. Also error messages will sometimes be printed if the operating system can figure out what the problem is. Typical of this class of problem is the message ATTEMPED DISC ALLOCATION FOR FILE SYS.PRT. CURRENT ALLOCATION FOR JOB EXCEEDS LIMIT FOR DISC RESERVATION TYPE HPT. INCREASE DISC LIMIT. For problems of this type, the solution is usually indicated, as in the above example.

The greatest part of this section will be devoted to solving user errors of the intermediate type, namely those that do not fall into the first category, but that can be solved with some detective work.

There are two principle sources of errors which can appear. First are the errors detected by the hardware, for example when ones attempts to divide by zero. Second are the software (program generated) errors. The latter usually come from either the ASC MATHPACK, or from ABEND requests generated by one of the language processors.

Of the latter type of error, namely those due to software procedures, the language translator ABEND's are solved simply by correcting the syntax of the program. The termination message, which will be found in the JSL summary, will be something like ABEND FLAG DETECTED, TERM = 10. For the other variety of software error, most commonly numerical errors which are detected by the system software, there is no handy way to pinpoint the source of troubles. An example of a numerical error would be an attempt to find the square root of a negative number. Although the associated traceback will indicate which routine was invoking the procedure, there is no handy means to find which line of code actually called the routine. Even if one knew, there might will be some indirect effect which caused this error in the first place. A procedure to find the actual line of code is discussed later and involves decoding the

base registers, B1 in particular. Numerical errors are most often caused by not understanding the limitations of the computer, the mathematics routines, or even sometimes an outright error, such as over writing data. The traceback which is printed is supposed to be a handy guide for finding the offending source code, but usually succeeds only in locating the routine where the offending data was found, and in many cases is rather obscure. The quickest way to find the source of this type of error is to put print statements in the program about where one thinks the errors occurs. This is not a very satisfactory procedure, but usually works.

The types of errors which are amenable to being solved on the spot are known as hardware errors and consist of

#1	DIVIDE CHECK - dividing by zero
#2	FLOATING POINT OVERFLOW - exceeding the upper limit of the floating word characteristic ($\simeq 73$)
#3	FLOATING POINT UNDERFLOW - exceeding the lower limit of the floating word characteristic ($\simeq -72$)
#4	FIXED POINT OVERFLOW - an integer operation which yields a number larger than $10^{31}-1$
#5	PROTECTION VIOLATION - accessing outside a user's area
#6	ILLEGAL INSTRUCTION - very unusual and will not be discussed here
#7	TIME OUT - no explanation needed

There is a mask which determines which of the arithmetic errors (#1—#4) will be trapped by the arithmetic unit and which will be ignored. It is set either by the user at the time the program is translated (compiled) or when the program is executing. This mask is called the ARITHMETIC EXECPTION MASK and is set by the C or U options when using the FORTRAN compilers. Assembly language users have to set it themselves. A couple of examples will probably be sufficient to illustrate how to find the offending source code which yields one of these errors. Errors of the type #5 or #6 are generally not under programmer control, but can be debugged in a similar fashion.

There are several debugging routines available to help find the cause of the above errors, but the most useful and easiest to understand is called COMMON. It is on the Scientific Program Library under the guise of debugging routines. No special effort is needed to obtain it except to CALL COMMON. Other aliases are MFDUMP, MFWIC, and HUMPTY (DUMPTY).

One of the primary advantages of this routine over others is that it is protected against destruction by the user, so even in cases where a complete over write of central memory occurs, some information should be returned. The examples which are given below will use this routine for purposes of explanation.

To illustrate the use of a traceback, we will use the following FORTRAN example

```
-------------------------------------------------------
            PROGRAM BLECH
            COMMON NX,NY,DX,A(100)
            CALL COMMON
            DX = 0.0
            DO 100 I=1, 100
    100     A(I) = FLOAT (I)
            X = 0.0
            DO 101 I = 1, 100
    101     X = X + 1.
            DO 102 I = 1, 100
    102     A(I) = A(I)/DX
            WRITE (6, 103) A, DX
    103     FORMAT (IX, 1P10G12.3)
            STOP
            END
-------------------------------------------------------
```

The first example will be one of a trackback when the FX compiler has been invoked and the second example will be with the NX compiler at level "K", the vectorization level.

Figure (3) shows the output from the mini-dump generated by COMMON. The **P**rogram **S**tatus **W**ord points to location (441). As mentioned earlier, this will usually be ahead of the actual location of the error although branch instructions can modify the sequence. Since the error is a DIVIDE CHECK, one should look for instructions which involve floating point arithmetic, and in particular, divide instructions. One occurs at absolute location (445), or relative location (045). This latter information is available if the macro MLNK has been used. The name of the routine is BLECH (as expected since we had only the one main program). Looking at the CSN/LOCATION listing produced by MFTN, Figure (4), we can see that this corresponds to CSN 11. This can be checked for this particular problem since the FX compiler was used. Once again, examining the mini-dump, one can see that CSN 11 was also given by the FX compiler. As expected, this refers to

102 A(I) = A(I)/DX.

A test using this same example with the NX compiler will demonstrate this same technique for vector parameter files (VPF) although in this case the correct CSN will not be directly available since the NX compiler does not provide this information. Figure (5) shows an equivalent mini-dump of the example program. Two VECT instructions appear at or prior to the PSW location and thus are candidates for the fault. Examination of the offset - see Appendix II - reveals that the offsets (relative to the READ ONLY area) are (20) and (28). Figure (6) shows the VPF's for this program. The offset is given in the upper left hand corner, as shown in Tables (III) and (IV). The VPF which corresponds to the floating divide is the second one and this is generated at CSN. Refering to the compilation in Figure (7), we see that, once again, the divide operaiton is found to be the culprit.

One final example, to show the effect of a PV or protection violation, will be given.

```
COMMON/DATA/NX,NY
CALL COMMON
NX = 1000 000
CALL SUBA(X)
STOP
END
SUBROUTINE SUBA(X,Y)
COMMON/DATA/NX,NY
REAL X(1)
Y = X(NX)
RETURN
END
```

The above example generated the mini-dump shown in Figure (8). Going through the same procdure, as before, we see that the PV was caused by the instruction

L A0 (000, BF), X4.

To confirm that it was the Y = X(NX) instruction which caused this problem the offset is (028) from the base register B2, and it is indexed by index register X1. X1 contains the index of 1000 000 which is clearly outside the program's area.

This should be a useful start. On final point which will help in reading these trackbacks in that, in general

B1 == subroutine return address (generated by a BLB)

B2 == READ/WRITE address

B3 == READ ONLY address

B4-B5 == Address of FIO routine if FIO is used

B6-B7 == are common areas

B8-B15 == are branch addressses or common areas as necessary

The above "B's" refer to base registers in the register dump portion of the mini-dump.

IX. Acknowledgment

Many thanks are due to those who participated in making this guide possible. E. McDonald originated the idea of compressed output which formed the basis of the modules "MFTN and MLNK". S. J. Marsh implemented these in both the macro and load modules. N. K. Winsor participated in the development of the CATMAN macro and load modules. Finally, there are many new users who have helped to improve this guide by actually trying the examples and having stumbled over parts which were not clear, have helped to make this a much more useful guide. This work was supported under NRL Task #SSPO/1781319.3083 and ONR Task #RR04-09-41.

APPENDIX I. - EXPLANATION OF MACROS

JSL statements are used to specify what action the operating system is to take for each part of the job stream. If one were to use the JSL only, he would find that certain combinations occurred repeatedly. Such groups of JSL can be combined and put in a form, with its own name, that is similar to, and will be processed just as if it were a JSL verb. This combination of JSL will be processed each time the given name is used. Such a structure is called a MACRO.

The form of a call which invokes a macro is identical to that for a JSL verb, and in fact, the same translator and syntax checker is used for both.

There is a set of macros which the system maintains for all users (a list of these is found in Section IV). The user may also maintain his own set of macros to perform special functions which are relevant to his own environment. In addition to the system macros, which come from the macro file whose access name is SYS.MCRO, there is a useful set of macros at the node

USERCAT/D77/L50/MACROS

this includes the macros MFTN, MLNK, MFXQT, ASG, GET, PUT, OUTPUT and CAT-MAN. To use these macros, use the following MACASG card

/ MACASG MACRX $$, USERCAT/D77/L50/MACROS

and make the following replacements

FTN → MFTN
LNK → MLNK
FXQT → MFXQT.

The remaining macros have no system equivalent, so that they are used as is. The individual macros and how they are used is explained below. The purpose of these macros is to provide the user more information, in a compact form (to save paper and storage), without undue burden on the user. Also, these macros have as defaults the options which are most commonly used, thus saving the beginning user from having to learn the entire system prior to using it.

MFTN MACRO

The purpose of the MFTN macro, which is used to invoke the FORTRAN compiler, provides just a little bit more information than provided by the standard FTN macro. This makes debugging a program very much simpler and faster. The cost is about 10% more than the stan-

dard compile without the "O" option. The "O" option is required if options are selected as in

FTNOPT=(........,O)

and the default is

FTNOPT=(K,M,U,V,D,Y,O).

There are six noteworthy points:

(1) It yields a similar listing to FTN, but also includes a summary, at the end of the source listing, of CSN versus relative core location in the program.

(2) In addition, the vector parameter files are printed at the end of the source listing. In finding errors in vector instructions, this shows exactly what variables are being used when the computer quits in the middle of a vector operation. When an error occurs during a vector operation, the particular vector being used can be found by using the traceback routine COMMON. The form of the vector parameter file and what each of the entries is used for is explained in Appendix II.

(3) It modifies SYS.OMOD to identify the compiler and compiler options used and places this information in the "I" card of the object module. When a program is linked, this information shows up in the link map.

(4) It allows the user to select, rather easily, his favorite compiler.

(5) Allows the user to use a non-standard list file (LIST = ...) without having to give FD cards.

(6) Allows the user to branch to specific parts of the job stream on the basis of compiler error termination codes (e.g. TERM = 8).

The various options for the MFTN macro are shown in Table (I) with the default, if there is one, shown in parenthesis.

The primary differences between the MFTN macro and the FTN macro are the set of three keywords: COMPILER, DOCUMENT and SAVE and a different choice of compiler defaults. The first one, COMPILER allows one to choose the favorite compiler. Note, however, the interaction with the keyword FTVERS, which overrides this option. The second keyword, DOCUMENT, which is also available as the first positional parameter, yields a short list of the options and their defaults. The third option, namely SAVE=YES or SAVE=NO, will

save the original object deck in the unmodified form at the access name SYS.NMOD. The defaults for FTN are FTNOPT = (K,M,O) and for MFTN, FTNOPT = (K,M,N,V,D,Y,O).

The only known problem is that when too little disk space is provided for the list file in the MFTN step, the routine WRITFL will timeout. The solution is to provide enough space. Note that the LIMIT card as well as LISTSIZE play a role in disk space determination.

The compilers which are currently available (2/12/79) are NX0509, NX0514, NX0522, NX0526, NX0527, NX, NEWNX, FX0518, FX0526, FX and NEWFX. The current compilers are NX and FX and experimental versions, if they exist, are called NEWNX and NEWFX.

MLNK MACRO

The purpose of the MLNK macro, which is used to invoke the linkage editor, is threefold: firstly, it provides all of the normal functions of the linkage editor as in LNK; secondly, the output, SYS.PRT, is much more compact and readable. If additional information is ever needed, it is a simple matter to execute a second LNK; thirdly, the intermediate file LNK.PRT. provides useful information to the debugging routine COMMON to yield relative program addresses and special diagnostics if a program fails while in a system routine. The parameters are shown in Table (II), with their defaults shown in parenthesis.

MFXQT MACRO

The MFXQT macro is a similarity extension of the two preceeding macros (MFTN and MLNK) and is identical to the FXQT macro except that it manipulates the files LNK.PRT., FT98F097 and FT98F098 for use by the debugging routine COMMON. If this routine is not used, or no error occurs, then the differences between these two macros are negligible. All options and keywords are the same as for FXQT.

If one wishes to duplicate the functions of this macro using the FXQT macro, the first order of business would be to FOSYS the file FT98F098 after the XQT verb was processed, i.e.,

```
/ FXQT
/ FOSYS FT98F098
```

The second iteration introduces the subtlety of defining FT98F097 and renaming LNK.PRT. to this access name. It is simpler, however, to use MFXQT in place of FXQT.

ASG MACRO

The JSL verb ASG requires the specification of the USE parameter, with the default being USE = EXC - exclusive. Since most users prefer the other option, namely USE = SHR - share, a macro is used to replace this JSL verb. It is identical to the verb itself, except the default is SHR, so the user need not type this extra bit of informaiton each time an assign command is done. It must be emphasized, however that this macro resides only at the L50/MACROS node and is **not** part of the system macro file.

CATMAN MACRO

This is a macro and associated program to generate the necessary JSL to manipulate cataloged files. Its most useful feature is the simplest, and that is to provide a readable catalog listing. The only required parameter is the path, which must be included completely, with no substitutions by PD's. Using the earlier example, we have

/ CATMAN USERCAT/Dmm/Bmm/USERC1 [,FOSYS = YES].

This would list all of a users files (with certain restrictions which are found in the R.C.C. computer note #160). The optional parameter, FOSYS = YES, is used if one wants to obtain the original catalog listing. Doing this **once** and comparing the two print-outs is most instructive.

APPENDIX II. - VECTOR PARAMETER FILES

A vector is a single instruction which is designed to perform one operation on a large mass of data. There are two advantages to using vector instructions for data whenever possible. The first is that less time is spent by the hardware in decoding instructions, since it only has to be done once for all of the data, whereas scalar code requires instruction interpretation of each piece of data. Since there is some overhead associated with loading a vector parameter file (VPF), vectors of length two are not more efficient than equivalent scaler code. The breakeven point turns out to be for data structures of length five of longer. The second advantage is that

entire DO-LOOPS can be contained on one instruction, and since the hardware can anticipate where the next data will be coming from, it can proceed to fetch it before the central procesor is ready. With scaler instructions, the memory buffer unit must await instruction decoding before it goes to fetch the data. Thus there is a two-fold loss in time. In general, vector instructions are designed to make vector and matrix arithmetic look like the straightforward operation that it is supposed to be.

In order to use a vector, the VPF must be loaded into V0-V7, the eight vector registers, and the operation started. With the two arithmetic units available on ASC(#7), two vector instructions can be executing together.

A vector parameter file consists of eight words and it is stored in a program's READ ONLY or READ/EXECUTE area, depending on the compiler which is used. When a vector is to be executed, the VPF is loaded and the command excuted using the instructions

```
L 5, (nnn, B3)
VECTX
```

or

```
VECTL (nnn, B3)
```

where nnn represents the offset in the READ ONLY area.

The form of a vector parameter file (VPF) is shown in Table (III). This is the form which appears in the source listing when MFTN is used, or the object listing wh n FTN is used. The purpose in providing this table is for decoding a VPF when debugging. Users might also want such a table when optimizing a program. An example of a VPF found in a source listing is shown in Table IV.

More information on the vector parameter files can be found in the "Programmers' Guide to the Central Processor", whose number is (#930039).

TABLE I

LABEL	OPERATION	OPERANDS
/[SYMBOL]	MFTN	[HELP] - THIS IS THE DOCUMENT LIST OPTION [, IN = ACCESS NAME (SYS.FIN)] [,OBJ = ACCESS NAME (SYS.OMOD)] [,FTNOPT = COMPILEROPTIONS (K,M,U,V,D,Y,O)] [,FTNTIME = CPTIME (30000)] [,FADDMEN = ADDMEN (7K)] [,SPACE = SPACE] [,VSPACE = VSPACE] [,OBJFILE = MOD or NEW(MOD)] [,OBJSIZE = BAND ALLOCATION (1/20/1)] [,LISTSIZE = BAND ALLOCATION (1/40/1)] [,LISTFILE = MOD OR NEW - (MOD) FOR BATCH AND - (NEW) FOR KEYBOARD USERS] [,NPIPES = INTEGER (2)] [,COMPILER = NXMMMM OR FXMMMM (NX0527)] [,FTVERS = COMPILER TYPE = NX OR FX (NX)] [,VPF = YES OR NO(YES)] [,MFTNSEC = TIME IN SECONDS (15)] [,SAVE = YES OR NO(NO)] [,LONGLIST = OBJECT LISTING DISPOSITION = REL, YES, NO (REL)]

TABLE II

LABEL	OPERATION	OPERANDS
/[SYMBOL]	MLNK	[DOCUMENT = YES OR NO(NO)] [,CONTROL = CONTROL CARDS (SYS.LEIN)] [,LIST = OUTPUT FILE (SYS.PRT)] [,OBJ = INPUT OBJECT FILE (SYS.OMOD)] [,LOAD = OUTPUT LOAD MODULE (SYS.LMOD)] [,LNKOPT = OP OPTIONS (M,A,X,F,Q,S)] [,START = STARTING ADDRESS - ENTRY POINT] [,NAME = NAME OF LOAD MODULE(GO)] [,LNKTIME = CPTIME FOR LINK (30000)] [,LADDMEN = ADDMEM (30K)] [,MEMLOAD = MEMBER NAME OF LOAD MODULE] [,LOADLIB = ACCESS NAME OF A PDS FILE] [,EOJ = LABEL] [,LSPACE = INTEGER WORDS ()] [,LISTFILE = NEW OR MOD (NEW)] [,OLIBPATH = OBJ LIBRARY FILE (SYSMOPL)] [,OLIBVERS = OBJ LIBRARY VERSION (+0)] [,SYSIN = YES OR NO(NO)] [,TABLDUMP = YES OR NO] [,RTP = RUN TIME PARAMETERS] [,MLNKSEC = MLNK RUN TIME (10)] [,LISTOPT = LIST OPTIONS FOR MLNK = C,M,R,E or U(C)]

TABLE III

ADDRESS	VRO	OPR	OPTIONS, (ALCT, SV, LEN)
	VR1	VCTRA	ADDRESS OF FIRST ARGUEMENT (SSA)
CSN	VR2	VCTRA	ADDRESS OF THE SECOND ARGUMENT (SSB)
	VR3	VCTRA	ADDRESS OF THE THIRD ARGUMENT (SSC)
	VR4	DATAH	INNER LOOP INCREMENT FOR ARG 1,2
	VR5	DATAH	INNER LOOP INCREMENT FOR ARG3 AND INNER LOOP COUNT
	VR6	DATAH	OUTER LOOP INCREMENT FOR ARG 1,2
	VR7	DATAH	OUTER LOOP INCREMENT FOR ARG 3 AND OUTER LOOP COUNT

ADDRESS = OFFSET RELATIVE TO THE "READ-ONLY" AREA
CSN = COMPILER STATEMENT NUMBER
VRO - VR7 = VECTOR REGISTER 0 THRU 7 (FOR A TOTAL OF EIGHT)
VCTRA = VECTOR ADDRESS
DATAH = DATA (TO BE GIVEN IN HALFWORDS)

VRO	OPR = OPERATION (E.G. VOR)
	ALCT = COMPARISON MASK
	SV = SINGLE-VALUED VECTOR SWITCH (SEE PAGE 9 of #93)
	LEN = LENGTH OF THE SELF LOOP
VR1	ADDRESS OF VECTOR "A" IN THE FORM (NNN,X REGISTER)
VR2	ADDRESS OF VECTOR "B" IN THE FORM (NNN,X REGISTER)
VR3	ADDRESS OF VECTOR "C" IN THE FORM (NNN,X REGISTER)
VR4	INNER LOOP INCREMENT (EACH IS A HALFWORD)
VR5	
VR6	OUTER LOOP INCREMENT (EACH IS A HALFWORD)
VR7	

TABLE IV

LOCATION	HEX VALUE	MNEMONIC	
000020	6E0D000A	VMF	0,10,13
	00000618	VCTRA	Y
6	00000228	VCTRA	Z
	00000230	VCTRA	X
	00010000	DATAH	1,0
	00010064	DATAH	1,100
	00000000	DATAH	0,0
	00000001	DATAH	0,1

which came from the statement

```
      DO 1 J = 1,100
      DO 1 I = 1,10
0006 1 X(I,J) = Z * Y(I,J)
```

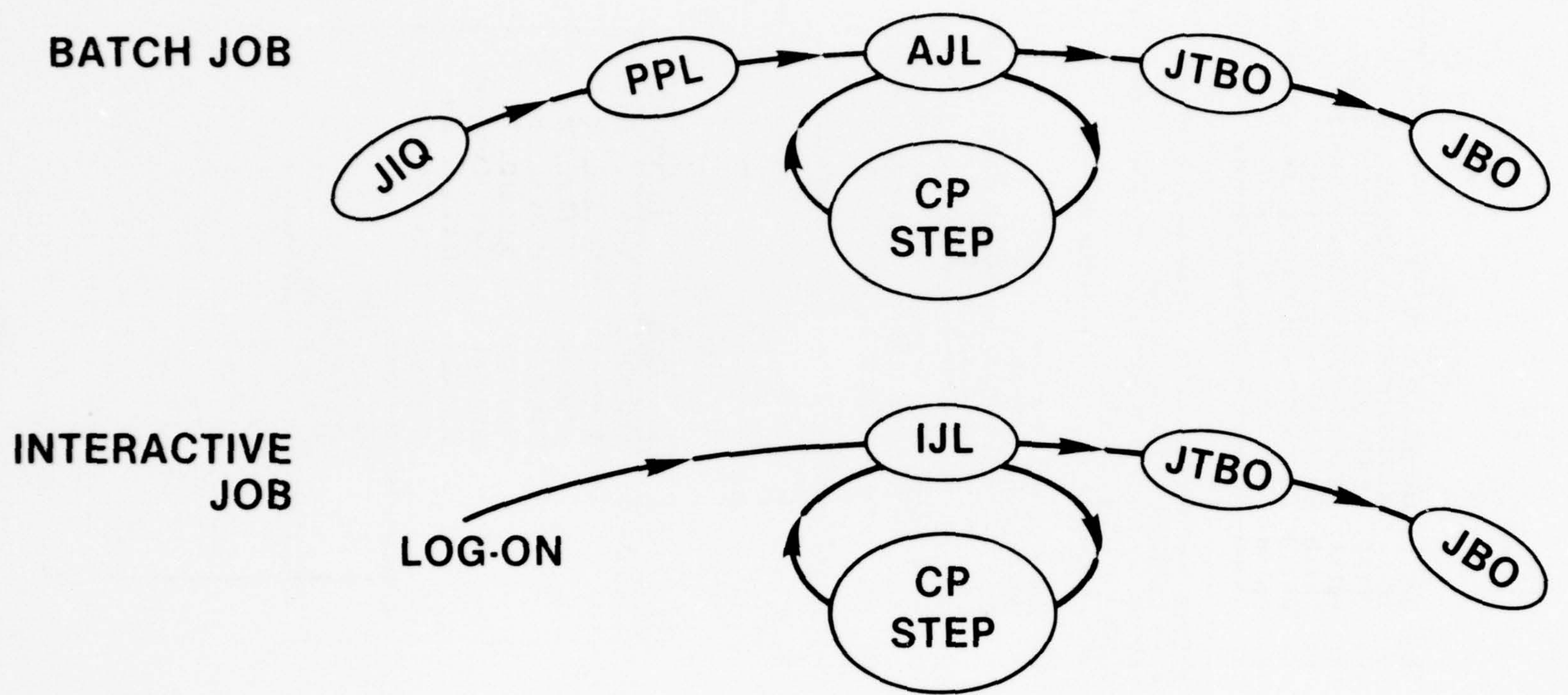


Fig. 1 — Schematic interaction of job blocks for both batch and interactive users

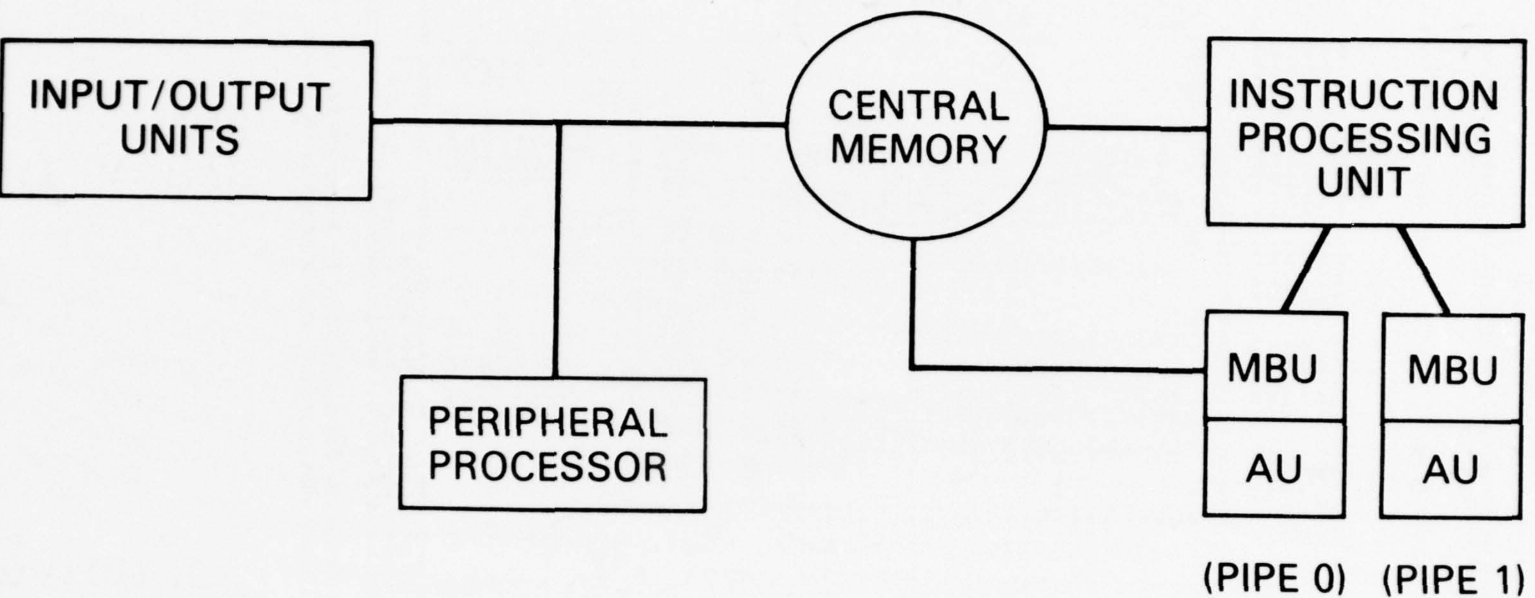


Fig. 2 — Components of the ASC mainframe

```
POST MORTEM DUMP FOR JOB #64BF, WHOSE JOB NAME IS WWJ80302     WHICH RAN AT 18:57:20  ON 03/02/79

THE MEMORY BUFFER UNIT THINKS THE JOB ABENDED AT MEMORY ADDRESS: 00000441  <----
THE NEXT TWO MEMORY REFERENCES WOULD BE TO:                      00000442
                                       AND:                      00000443

THE PROGRAM STATUS WORD AT TERMINATION WAS:                      AE03C012  THE ERROR WAS  DIVIDE CHECK

                                  0/8       1/9       2/A       3/B       4/C       5/D       6/E       7/F

ERROR IN    BLECH             B0 00000000  0E000413  00001800  00000600  00001A68  00001A80  00001A88  00000000
                              B8 00000000  00000000  00000000  00000000  00000000  00000000  00005E58  00000466
                              A0 00000001  00000064  00000000  00000000  00000000  00000000  00000000  00000000
(CSN 11 IF "FX" COMPILER)     A8 00000000  00000000  00000000  00000000  00000000  00000000  00000001  00000064
                              X0 00000000  00000008  00000000  00000000  00000001  00000001  00000001  00000003
                              V0 00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000

                              DECODE OF INSTRUCTIONS IN CENTRAL MEMORY

                              LOC     CODE       OP      ARGS                RELATIVE ADDRESS
                              430     16000008   LLA     0   (08)                  0030
                              431     2C504010   ST      X5  (010,B4)              0031
                              432     5C100009   LI      X1  0009                  0032
                              433     14004011   L       A0  (011,B4)              0033
                              434     42003012   AF      A0  (012,B3)              0034
                              435     24004011   ST      A0  (011,B4)              0035
                              436     1C504010   L       X5  (010,B4)              0036
                              437     1700223C   LD      A0  (23C,B2)              0037
                              438     86500FF8   BCLE    X5  (F8)                  0038
                              439     5C10000A   LI      X1  000A                  0039
                              43A     54E00001   LI      AE  0001                  003A
                              43B     54F00064   LI      AF  0064                  003B
                              43C     27E0223C   STD     AE  (23C,B2)              003C
                              43D     5C500001   LI      X5  0001                  003D
                              43E     00000000   IND         00000000              003E
                              43F     00000000   IND         00000000              003F
                              440     1600000A   LLA     0   (0A)                  0040
                              441     2C504010   ST      X5  (010,B4)              0041
                              442     5C10000B   LI      X1  000B                  0042
                              443     1C404010   L       X4  (010,B4)              0043
                              444     14046002   L       A0  (002,B6),X4           0044
                       ---->  445     66006002   DF      A0  (002,B6)              0045
                              446     1C404010   L       X4  (010,B4)              0046
                              447     24046002   ST      A0  (002,B6),X4           0047
                              448     1C504010   L       X5  (010,B4)              0048

                                    MEMORY BUFFER ADDRESSES

                              PIPE      X          Y          ZP         Z

                                1   A2001A88   0A001A38   00001A88   A2001A38
                                2   18001A78   1C000610   0A001A78   02001A88

                                    DECODE OF INSTRUCTIONS IN THE INSTRUCTION BUFFER

                         BUFFER "A"                                      BUFFER "B"
            LOC     CODE       OP      ARGS                 LOC     CODE       OP      ARGS
             0      1600000A   LLA     0   (0A)              0      1C504010   L       X5  (010,B4)
             1      2C504010   ST      X5  (010,B4)          1      1700223C   LD      A0  (23C,B2)
             2      5C10000B   LI      X1  000B              2      86500FF6   BCLE    X5  (F6)
             3      1C404010   L       X4  (010,B4)          3      5C10000C   LI      X1  000C
             4      14046002   L       A0  (002,B6),X4       4      2000204F   STZ         (04F,B2)
             5      66006002   DF      A0  (002,B6)          5      5C800006   LI      V0  0006
             6      1C404010   L       X4  (010,B4)          6      56904008   LEA     V1  (008,B4)
             7      24046002   ST      A0  (002,B6),X4       7      5CA00000   LI      V2  0000

ASSEMBLY LANGUAGE MNEMONICS CAN BE FOUND IN THE "PROGRAMMERS GUIDE TO THE CENTRAL PROCESSOR #930039"
BASE REGISTER ONE (B1) CONTAINS THE SUBROUTINE RETURN ADDRESS
BASE REGISTER TWO (B2) CONTAINS THE ADDRESS OF THE PROGRAM DATA AREA
INDEX REGISTER ONE (X1) HOLDS THE CSN NUMBER FOR PROGRAMS COMPILED WITH THE "FX" COMPILER
```

Fig. 3 — Mini-dump generated by COMMON. The first example (see text) generated this dump when the program was compiled with the FX compiler.

```
                 SOURCE LISTING            ASC FAST FORTRAN COMPILER     RELEASE FTFX0526.P214          PAGE 0001
   CSN              STATEMENT              CP OPTIONS = (D,K,M,B,U,V,Y)  DATE = 03/02/79(79.061)     TIME = 18:55:34

   0001             PROGRAM BLECH
   0002             COMMON NX,NY,DX,A(100)
   0003             CALL COMMON
   0004             DX = 0.0
   0005             DO 100 I = 1, 100
   0006      100    A(I) = FLOAT(I)
   0007             X = 0.0
   0008             DO 101 I = 1, 100
   0009      101    X = X + 1.
   0010             DO 102 I = 1, 100
   0011      102    A(I) = A(I) / DX
   0012             WRITE(6,103) A, DX
   0013      103    FORMAT(1X,1P10G12.3)
   0014             STOP
   0015             END


          BLECH                            OBJECT CODE SUMMARY

 CSN  LOC       CSN  LOC       CSN  LOC       CSN  LOC       CSN  LOC

      0000
 0001 0008
 0003 000C
      000D
 0004 0013
      0014
 0005 0016
      0017
 0006 001D
      001E
 0007 0025
      0026
 0008 0027
      0028
 0009 0032
      0033
 0010 0039
      003A
 0011 0042
      0043   <----
 0012 004B
      004C
 0014 0058
      0059

SPACE UTILIZATION SUMMARY

 STA NUM    LOCATION   INSTRUCTION        LABEL         OPCODE  OPERAND              BCD OPERAND

  6024 WORDS ALLOCATED FOR BUFFERS
   251 WORDS ALLOCATED FOR TABLES
     0 ADDITIONAL WORDS REQUIRED FOR XREF
 13725 WORDS REMAIN UNUSED

TOTAL ALLOCATION =    20000 WORDS

          0  **W** ERRORS DETECTED
          0  **E** ERRORS DETECTED
          0  **C** ERRORS DETECTED
```

Fig. 4 — Sample program compiled with the FX compiler. This is the example used to generate the post-mortem dump shown in Fig. (3).

```
POST MORTEM DUMP FOR JOB #64BF, WHOSE JOB NAME IS WWJS0302     WHICH RAN AT 18:59:51  ON 03/02/79

THE MEMORY BUFFER UNIT THINKS THE JOB ABENDED AT MEMORY ADDRESS: 00000427  <--
THE NEXT TWO MEMORY REFERENCES WOULD BE TO:                      00000428
                                        AND:                     00000429

THE PROGRAM STATUS WORD AT TERMINATION WAS:                      AE03C011  THE ERROR WAS  DIVIDE CHECK

                                  0/8       1/9       2/A       3/B       4/C       5/D       6/E       7/F

ERROR IN     BLECH             B0 00000000  0E00041A  00001A00  00001900  00005E58  00001C38  00000000  00000000
                               B8 00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
                               A0 41100000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
(CSN*** IF "FX" COMPILER)      A8 00000000  00000000  00000000  00000000  00000000  00000000  00000000  42640000
                               X0 00000000  00001A44  00000000  00000000  00000000  00000000  00000000  00000000
                               V0 660D0032  00001C6D  00001C3A  00001C6D  00000000  00000001  00000000  00000001

                               DECODE OF INSTRUCTIONS IN CENTRAL MEMORY

                               LOC    CODE      OP     ARGS            RELATIVE ADDRESS
                               418    56803018  LEA    V0   (018,B3)        0018
                               419    98180001  BLB    B1 I (01)            0019
                               41A    000004C8  IND         000004C8        001A
                               41B    20002231  STZ         (231,B2)        001B
                               41C    1400301B  L      A0   (01B,B3)        001C
                               41D    5C800064  LI     V0   0064            001D
                               41E    00000000  IND         00000000        001E
                               41F    00000000  IND         00000000        001F
                               420    16000004  LLA     0   (04)            0020
                               421    14F02231  L      AF   (231,B2)        0021
                               422    42F00010  AF     AF   (10)            0022
                               423    24F02231  ST     AF   (231,B2)        0023
                               424    8F800FFC  DBNZ   V0   (FC)            0024
                        ---->  425    80203020  VECT    2   (020,B3)        0025
                        ---->  426    80403028  VECT    4   (028,B3)        0026
                               427    5CF0000C  LI     V7   000C            0027
                               428    5CA00000  LI     V2   0000            0028
                               429    5CB00000  LI     V3   0000            0029
                               42A    56902038  LEA    V1   (038,B2)        002A
                               42B    2CF02003  ST     V7   (003,B2)        002B
                               42C    5C800006  LI     V0   0006            002C
                               42D    98104002  BLB    B1   (002,B4)        002D
                               42E    5C900064  LI     V1   0064            002E
                               42F    56805003  LEA    V0   (003,B5)        002F
                               430    9810400F  BLB    B1   (00F,B4)        0030

                                    MEMORY BUFFER ADDRESSES

                               PIPE     X         Y         ZP        Z

                                 1  00001C30  00001918  00001C30  00001C30
                                 2  88001918  8C001918  0A001A00  02001C38

                                    DECODE OF INSTRUCTIONS IN THE INSTRUCTION BUFFER

                    BUFFER "A"                                          BUFFER "B"
          LOC     CODE      OP      ARGS                    LOC     CODE      OP      ARGS
           0    5CA00000    LI      V2   0000                0    9810400F    BLB     B1   (00F,B4)
           1    5CB00000    LI      V3   0000                1    56805002    LEA     V0   (002,B5)
           2    56902038    LEA     V1   (038,B2)            2    98104008    BLB     B1   (008,B4)
           3    2CF02003    ST      V7   (003,B2)            3    98104004    BLB     B1   (004,B4)
           4    5C800006    LI      V0   0006                4    5C80FFFF    LI      V0   0FFF
           5    98104002    BLB     B1   (002,B4)            5    98104001    BLB     B1   (001,B4)
           6    5C900064    LI      V1   0064                6    7F000000    ***
           7    56805003    LEA     V0   (003,B5)            7    7F000000    ***

ASSEMBLY LANGUAGE MNEMONICS CAN BE FOUND IN THE "PROGRAMMERS GUIDE TO THE CENTRAL PROCESSOR #030039"
BASE REGISTER ONE (B1) CONTAINS THE SUBROUTINE RETURN ADDRESS
BASE REGISTER TWO (B2) CONTAINS THE ADDRESS OF THE PROGRAM DATA AREA
INDEX REGISTER ONE (X1) HOLDS THE CSN NUMBER FOR PROGRAMS COMPILED WITH THE "FX" COMPILER
```

Fig. 5 — Mini-dump generated by COMMON. The first example(see text) generated this dump when the program was compiled with the NX compiler.

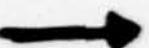
```
BLECH                                  VECTOR PARAMETER FILES                                PAGE    6

LOCATION HEX VALUE MNEMONIC            LOCATION HEX VALUE MNEMONIC            LOCATION HEX VALUE MNEMONIC
 (CSN)                                  (CSN)                                  (CSN)

000020  660D0032  VDF    0,50,13       000028  660D0032  VDF    0,50,13
        00000003  VCTRA  A                     00000035  VCTPA  A+#032
  11    00000002  VCTRA  DX              11    00000002  VCTPA  DX
        00000003  VCTRA  A                     00000035  VCTPA  A+#032
        00000000  DATAH  0,0                   00000000  DATAH  0,0
        00000001  DATAH  0,1                   00000001  DATAH  0,1
        00000000  DATAH  0,0                   00000000  DATAH  0,0
        00000001  DATAH  0,1                   00000001  DATAH  0,1

      0  **W** ERRORS DETECTED
      0  **E** ERRORS DETECTED
      0  **C** ERRORS DETECTED
```

Fig. 6 – Vector Parameter Files generated for the first debugging example (see text) when the program was compiled with the NX (level K) compiler.



```
                    SOURCE LISTING            ASC NX FORTRAN COMPILER        RELEASE FCNX05.027.139          PAGE  0001
        CSN              STATEMENT            CP OPTIONS = (D,K,M,O,U,V,Y)   DATE = 03/02/79(79.061)     TIME = 18:57:53

        0001             PROGRAM BLECH
        0002             COMMON NX,NY,DX,A(100)
        0003             CALL COMMON
        0004             DX = 0.0
        0005             DO 100 I = 1, 100
        0006      100    A(I) = FLOAT(I)
        0007             X = 0.0
        0008             DO 101 I = 1, 100
        0009      101    X = X + 1.
        0010             DO 102 I = 1, 100
  -->   0011      102    A(I) = A(I) / DX
        0012             WRITE(6,103) A, DX
        0013      103    FORMAT(1X,1P10G12.3)
        0014             STOP
        0015             END


                  BLECH                              OBJECT CODE SUMMARY

      CSN    LOC        CSN    LOC        CSN    LOC        CSN    LOC        CSN    LOC

             0000
      0003   0008
      0006   0009
      0003   000A
      0006   000D
      0003   000F
             0010
      0006   0011
      0004   0012
      0006   0013
             0015
      0006   0016
             001A
      0007   001B
             001C
      0009   0021
      0011   0025
      0012   0027
      0014   0034
```

Fig. 7 — Listing of the program used to generate the protection violation

```
POST MORTEM DUMP FOR JOB #89CD, WHOSE JOB NAME IS WWJS0315     WHICH RAN AT 16:41:45  ON 03/15/79

THE MEMORY BUFFER UNIT THINKS THE JOB ABENDED AT MEMORY ADDRESS:  00000442  <---
THE NEXT TWO MEMORY REFERENCES WOULD BE TO:                       00000443
                                     AND:                         00000444

THE PROGRAM STATUS WORD AT TERMINATION WAS:                       0EC3C012  THE ERROR WAS A PROTECTION VIOLATION

                                   0/8       1/9       2/A       3/B       4/C       5/D       6/E       7/F

ERROR IN    SUBA               B0 00000000  00001805  00001A88  00000618  00001AF0  00001808  00001A80  00000000
                               B8 00000000  00000000  00000000  00000000  00000000  00000000  00005E58  00001A70
                               A0 000F4240  00000000  00000000  00000000  00000000  00000000  00000000  00000000
(CSN  4 IF "FX" COMPILER)      A8 00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
                               X0 00000000  00000004  00000000  0E00042A  000F423F  00001800  00000001  00000005
                               V0 00001A70  00000000  00000000  00000000  00000000  00000000  00000000  00000000

                               DECODE OF INSTRUCTIONS IN CENTRAL MEMORY

                               LOC      CODE      OP       ARGS                RELATIVE ADDRESS
                               430    99300002    BLX    X3   (02)                   0008
                               431    00000618    IND         00000618               0009
                               432    18060000    LF      0 I (00),X3                000A
                               433    18103008    LF      1   (008,B3)               000B
                               434    18F03001    L      BF   (001,B3)               000C
                               435    18E0F000    L      BE   (000,BF)               000D
                               436    7230FFF9    AI     X3   0FF9                   000E
                               437    2C302007    ST     X3   (007,B2)               000F
                               438    2C502000    ST     X5   (000,B2)               0010
                               439    5C600001    LI     X6   0001                   0011
                               43A    11002062    LEM     0   (062,B2)               0012
                               43B    2C802038    ST     V0   (038,B2)               0013
                               43C    5C100004    LI     X1   0004                   0014
                               43D    1C406000    L      X4   (000,B6)               0015
                               43E    18F02038    L      BF   (038,B2)               0016
                               43F    7240FFFF    AI     X4   0FFF                   0017
                          ---> 440    1404F000    L      A0   (000,BF),X4            0018
                               441    24004010    ST     A0   (010,B4)               0019
                               442    5C100005    LI     X1   0005                   001A
                               443    18F03000    L      BF   (000,B3)               001B
                               444    2000F00F    STZ         (00F,BF)               001C
                               445    5C800006    LI     V0   0006                   001D
                               446    56904008    LEA    V1   (008,B4)               001E
                               447    5CA00000    LI     V2   0000                   001F
                               448    5CB00000    LI     V3   0000                   0020

                                      MEMORY BUFFER ADDRESSES

                               PIPE      X          Y          ZP          Z

                                 1   C0001AC0   030F5CA8   08001B00   A4001AC0
                                 2   18001AE8   1C000610   0A001A88   02001800

                                      DECODE OF INSTRUCTIONS IN THE INSTRUCTION BUFFER

                     BUFFER "A"                                            BUFFER "B"
        LOC      CODE      OP        ARGS                     LOC      CODE      OP        ARGS
         0     1404F000    L      A0   (000,BF),X4              0    5CB00000    LI     V3   0000
         1     24004010    ST     A0   (010,B4)                 1    9810F002    BLB    B1   (002,BE)
         2     5C100005    LI     X1   0005                     2    56804010    LEA    V0   (010,B4)
         3     18F03000    L      BF   (000,B3)                 3    9810F008    BLB    B1   (008,BE)
         4     2000F00F    STZ         (00F,BF)                 4    9810F004    BLB    B1   (004,BE)
         5     5C800006    LI     V0   0006                     5    5C100007    LI     X1   0007
         6     56904008    LEA    V1   (008,B4)                 6    18102007    L      B1   (007,B2)
         7     5CA00000    LI     V2   0000                     7    99001000    BLX    X0   (000,B1)

ASSEMBLY LANGUAGE MNEMONICS CAN BE FOUND IN THE "PROGRAMMERS GUIDE TO THE CENTRAL PROCESSOR #930039"
BASE REGISTER ONE (B1) CONTAINS THE SUBROUTINE RETURN ADDRESS
BASE REGISTER TWO (B2) CONTAINS THE ADDRESS OF THE PROGRAM DATA AREA
INDEX REGISTER ONE (X1) HOLDS THE CSN NUMBER FOR PROGRAMS COMPILED WITH THE "FX" COMPILER
```

Fig. 8 — Mini-dump generated by COMMON which was called by the program shown in Fig. (7).

☆ U. S. GOVERNMENT PRINTING OFFICE : 1979 281-484/60