PIODRP, FLXDBG 4/15/93

Need to ① run tests of EDRLST, PIODRP, PFRTPL, PRNPHA, FLXDBG, FLXLST

on

Pioneer G EDR tapes
Pioneer F EDR tapes with format B, B/D format

② Rewrite BITZON (optional?)

③ Add a "catalog" type structure

Remove/Rewrite sections of code dealing with implementing catalog (ie merges etc)

END OF NOTES

~ Am 4/15
recent back up on dolphin
/home/dolphin/crick/piobak

after this backup, david was fixing up debug commands and last minute f+10 fixups relating to useage (such as check fopen code on return) etc.

no diskette backup of final source

John has CRICK password

final [see pg 13, hand notes)

ADAPTING PIONEER PRODUCTION SOFTWARE TO THE SUN[1]

INTRODUCTION

In the overall effort to move cosmic ray data production and analysis away from the IBM MVS environment, staff[2] have adapted Pioneer cosmic ray data production tools (EDRLST, PIODRP, PFRTPL, PRNPHA, FLXDBG, and FLXLST) to the Sun Unix environment. The original IBM source code to these tools is written in FORTRAN and IBM assembly language, and the adapted Sun source code in FORTRAN and C. This document presents some of the considerations taken in this conversion effort in the anticipation that future conversion efforts must deal with similar issues.

The essence of this document is a presentation both of the conditions under which program changes are necessitated, and of the suggested transformations to be applied to programs in conversion work. This is by no means an exhaustive work, but rather a summary of experiences gained from a single conversion effort.

CONVERSION GOALS AND GUIDING PRINCIPLES

The objective of the conversion task was to move software off the IBM. The two primary goals of the conversion effort were that the conversion effort be quick and that the conversion be correct in that it matches the IBM version. To achieve these goals, two primary principles were applied:

Principle 1	*Minimize changes in the original FORTRAN source code to only those changes which are absolutely necessary.*
Principle 2	*Converted assembly routines should be tested as thoroughly as possible against the existing IBM routines before use.*

The first principle was used to expedite the creation of an initial working version on the Sun. The motivation for this is that once the software is essentially working on the Sun, then that dependence on the IBM (with respect to conversion work) is broken. Additional development in adaptation work could then be localized to the Sun workstation environment. The advantage of this approach is that the adaptation work becomes less dependent on the existence of the IBM.

Expedition of assembly routine conversion is difficult to guarantee without formal tools and mechanisms. One technique employeed by staff when analyzing assembly routines was to describe the state of computation (formally or semi-formally) as logical formulations in terms of both the low level machine and the application domain. In other words, staff used assertions or identified invariant conditions at various points in the assembly routines describing changes to state of the machine. Of course, as there is no formal axiomatization of the IBM system, the axioms applied were based on the best understanding of the system. These assertions were then turned into program specifications that were used to derive converted routines using program synthesis techniques. In short, the applied conversion process implemented of a series of language transformations from assembly routines to low level assertions to application domain level assertions to specifications to converted routines.

---

[1]Written by David Crick (Hughes STX), April 16, 1993. Thanks to Nitya Nath, John Katen, Henry Lo, and Pamela Schuster (Hughes STX) for helpful comments.

[2]David Crick, John Katen, Henry Lo, Ramesh Ponneganti, and Pamela Schuster (Hughes STX).

An implication of the first principle is that extreme caution should be used when removing large sections of code (however safe it seems) and when changing types of variables (especially those in shared common blocks). This implied principle and the second principle above provided some protection against inadvertently introducing subtle bugs into the software, thus supported the maintenance of the correctness goal.

In achieving the correctness goal, testing had to be employed. At the routine or program block level, formal testing techniques were used, often resulting in exhaustive testing. The formal tests would, for example, test paths of execution or extreme cases in the routines. One approach applied in formalizing the testing of routines that depend on data read from files was to simulate the reading of live data files. The danger in testing routines only by reading actual live data files is that the tests then become dependent on the contents of the files, and certain extreme cases might be missed. In general, routines in question should be treated as independent units of which all the inputs are clearly understood and controllable by the tester.

At the higher system level, elaborate tests were conducted to guarantee that the Sun version saw everything the same as the IBM version. At this level, testing of live data files is essential. In general, testing the correctness of the conversion resulted in the greatest consumption of time.

When executing extensive tests, bugs may be discovered in the original IBM versions of software. Depending on the nature of the bugs and the format of the testing, in some cases it may not be wise to correct preexisting bugs on the Sun version until after testing has been completed. Typically this situation occurs when the IBM version with the bug has been run before the test, the test depends on the results of the run, and it is practically impossible to rerun a corrected IBM version (*e.g.*, an EDR tape may no longer exist or a catalog may be destroyed).

Once the initial conversion has been made and verified correct, then bug fixes and more significant changes to the code could be considered. At that point, different Sun versions could be compared against the initial Sun version. Hence, it is crucial that the verification of the initial conversion be of high quality.

Differences in FORTRAN Implementations

Several differences have been observed between the Sun and IBM FORTRAN implementations. Some of these differences are presented in this section.

Differences in Constant Expressions

On the IBM, Boolean constant expressions may be denoted by `T` and `F`. These must be changed to `.TRUE.` and `.FALSE.` on the Sun. Similary, hexadecimal constants may be denoted, for example, as `Z01FFFFFF` on the IBM, but must be changed to `X'01FFFFFF'` on the Sun. Finally, statement labels on the IBM may be referenced, for example, by `&1000`, but must be referenced by `*1000` on the Sun. So, given the following fragment of IBM source code

```
      IMPLICIT LOGICAL*1(Q)
      DATA Q/T/,I/Z01FFFFFF/
 1000 CALL TEST(&1000)
      .
      .
      .
      SUBROUTINE TEST(&)
```

the converted fragment should appear as

```
      IMPLICIT LOGICAL*1(Q)
      DATA Q/.true./,I/x'01FFFFFF'/
 1000 CALL TEST(*1000)
      .
      .
      .
      SUBROUTINE TEST(*)
```

Notice the use of lower case text in the changed code. This convention allows quick identification of changed code, and was used wherever possible in the Pioneer conversion.

Differences in Namelist Input

Affecting the Pioneer conversion, there are two significant differences in the way in which namelist input works. The first difference is that on the Sun, a logical variable whose value is input via a namelist must be of type LOGICAL*4 while on the IBM it may be of type LOGICAL*1. In this case, suppose there is an input variable Q of type LOGICAL*1. A new identifier Qnew of type LOGICAL*1 is introduced and all references to Q in that module are changed to Qnew (essentially renaming Q as Qnew). Then the type of Q is changed to LOGICAL*4, and Q=Qnew and Qnew=Q are inserted before and after, respectively, the namelist input. In this way we preserve the interface to the program (the input variable is still identified by Q), and the type and functionality of Q through Qnew (it may be the case that Q is in a shared common block, so it may be desirable to preserve the type of Q). For example, given the following fragment of IBM source code

```
      LOGICAL*1 Q1,Q2,Q3,Q4
      LOGICAL*4 Q5
      COMMON   /SHARE/ Q1,Q2,Q3,Q4,Q5
      NAMELIST /INPUT/ Q1,Q5
      Q1=.FALSE.
      Q5=.TRUE.
      READ(5,INPUT)
      IF (Q1.OR.Q5) STOP
```

the converted fragment should be

```
      LOGICAL*1 Q1new,Q2,Q3,Q4
      LOGICAL*4 Q1,Q5
      COMMON   /SHARE/ Q1new,Q2,Q3,Q4,Q5
      NAMELIST /INPUT/ Q1,Q5
      Q1new=.FALSE.
      Q5=.TRUE.
      q1=q1new
      READ(5,INPUT)
      q1new=q1
      IF (Q1new.OR.Q5) STOP
```

Again, this transformation preserves the name of the input variable, the type of the variable shared in the common block, and the functionality of the routine. The need for the two variables representing the same value arises essentially from the fact that the types of the namelist input variable and the shared variable differ.

The second difference is that on the IBM, character strings may be read via namelists into variables of type REAL*4, REAL*8, or COMPLEX*16, whereas on the Sun, they must be read in as variables of type CHARACTER*. The same approach as that above is taken except that, in this case, the two variables may be equivalenced. When the two variables have been equivalenced, uses of the variable of type CHARACTER* should be made wherever the variable is used as a character string. Also, since the two variables have been equivalenced, the need for copying before and after the namelist read is eliminated. As an example, the following fragment of IBM code

```
      IMPLICIT REAL*8(D)
      COMPLEX*16 C1
      CHARACTER*1 C2
      COMMON   /SHARE/ D1,D2,C1,C2
      NAMELIST /INPUT/ D1,D2,C1,C2
      DATA D1/'TEST'/,D2/10.0325/
      READ(5,INPUT)
      WRITE(6,10) D1,D2,C1,C2
   10 FORMAT(1X,A8,F9.4,A16,A1)
```

should be converted as

```
      IMPLICIT REAL*8(D)
      COMPLEX*16 C1new
      CHARACTER*1 C2
      character*8 d1
      character*16 c1
      equivalence (d1,d1new),(c1,c1new)
      COMMON   /SHARE/ D1new,D2,C1new,C2
      NAMELIST /INPUT/ D1,D2,C1,C2
      DATA D1/'TEST'/,D2/10.0325/
      READ(5,INPUT)
      WRITE(6,10) D1,D2,C1,C2
   10 FORMAT(1X,A8,F9.4,A16,A1)
```

Note that in this example, it would have been acceptable to keep the common line as

```
      COMMON   /SHARE/ D1,D2,C1,C2
```

thereby eliminating the need for `D1new` and `C1new` altogether. As stated earlier, care should be taken when changing the types of variables in shared common blocks.

Differences in Hexadecimal Formats, and Octal Formats

Hexadecimal output formats differ in the Sun and IBM implementations of FORTRAN. To enable writing on the Sun to match that on the IBM, two routines `hexfmt` and `octfmt` were written.

The subroutine call `hexfmt(i,c,l)` writes the integer `i` into the first `l` characters of the character string `c` in a hexadecimal format matching what the corresponding write on the IBM would produce. For example, the following fragment from `writer` (part of PFRTPL)

```
      WRITE(6,35)...,(ISPARE(I),I=1,2)
   35 FORMAT(2X,...,2(1X,Z8))
```

was converted as

```
      character*8 cspare(2)
      .
      .
      .
      call hexfmt(ispare(1),cspare(1),8)
      call hexfmt(ispare(2),cspare(2),8)
      WRITE(6,35)...,(cSPARE(I),I=1,2)
 35   FORMAT(2X,...,2(1X,a8))
```

The subroutine call `octfmt(i,c,l)` behaves the same as `hexfmt`, except that an octal format is used. This routine was used in the conversion of EDRLST.

Remnants from older versions of FORTRAN

Several sources of programs were written under older versions of FORTRAN. In general, care should be taken that hidden remnants from these earlier versions do not lie dormant in software. Here we list cite some of the occurrences of these types of problems.

The following construction appeared in EDRLST and was legal under the IBM interpretation of FORTRAN, but illegal under the Sun interpretation:

```
 42   .
      .
      .
      DO 175 NSC=1,3
      .
      .
      .
 80   <STATEMENT>
      GOTO 42
      .
      .
      .
175   CONTINUE
      .
      .
      .
      DO 610 NSC=1,5
      .
      .
      .
      IF (<TEST>) GO TO 80
      .
      .
      .
600
610   CONTINUE
```

Essentially this construction may be characterized as jumping from the inside of one loop to the inside of another. In general, careful analysis may be necessary to provide the proper transformation to remove such a construction. In this case in EDRLST, the following conversion worked:

```
 42   .
      .
      .
      DO 175 NSC=1,3
      .
      .
      .
 80   <STATEMENT>
      GOTO 42
      .
      .
      .
175   CONTINUE
      .
      .
      .
      DO 610 NSC=1,5
      .
      .
      .
      IF (<TEST>) GO TO 590
      .
      .
      .
      GO TO 600
590   <STATEMENT>
      GO TO 42
600
610   CONTINUE
```

Essentially the *jumped to* block was copied into the *jumped from* loop preserving the semantics of the program.

Another example of source code written in older versions of FORTRAN occurs in how the initial element of an array may be referenced in an equivalence declaration. The following is legal in some versions of FORTRAN, but is illegal on the Sun:

```
      DIMENSION A(100),B(10,10)
      EQUIVALENCE (A(1),B(1))
```

The updated version, of course, should be

```
      DIMENSION A(100),B(10,10)
      EQUIVALENCE (A(1),B(1,1))
```

DIFFERENCES IN LIBRARIES

In the Pioneer production source code, several calls to IBM system libraries are made. The essential IBM routines had to be adapted to the Sun as well. The most crucial of these, the *FTIO* library (including `fopen`, `fread`, `fwrite`, and `fclose`), was provided by NASA.[3] Two other referenced routines, `abend` and `fmove`, were converted by staff and require no additional special consideration. Discussed here are the FTIO considerations. Calls to the *DAIO* library were not used in the Sun version, and therefore not converted.

[3]Written by Nand Lal (GSFC) and E.S. Panduranga (STX), 1991.

The two primary concerns when using the FTIO routines on the Sun are that a file must be opened using `fopen` before reading from it or writing to it using `fread` or `fwrite`, and that, after use, a file should be closed using `fclose` if the file has been written to using `fwrite`. No special consideration need be taken regarding calls to `fread` and `fwrite`, with one exception that is discussed at the end of this section.

A possible indication of where an `fopen` should appear is where either a `mount` or a `posn` call is made in the IBM source code. For example, in the source code to the FLXDBG routine `dbgtap`, the following fragment

```
      CALL MOUNT2(2,NSUMUN,DTAPOT,1,&3999)
      .
      .
      .
      ENTRY MOUNT2(IOF,LUNO,DVLSER,NFILE,*)
      .
      .
      .
      CALL MOUNT(IOF,LUNO,DVLSER,NFILE)
```

indicates the need of a call to `fopen`. This was then converted to

```
      integer*4 fopen
      character*16 cfile
      .
      .
      .
      lentap=islen(ctapot)
      cfile=ctapot(1:lentap)//'.flx'
      if (fopen(2,nsumun,cfile,0,'VB',32760,32760).ne.0)
     1   call fopenerr(cfile)
```

Not shown here is the fact that the character string `ctapot` must have been earlier assigned a value serving as the prefix to a Unix file name. The utility functions `islen` and `fopenerr` were written to assist in this transformation. The function call `islen(c)` returns the length of the character string `c` up to trailing blanks, and the subroutine call `fopenerr(cfile)` reports a problem opening a file and aborts the program. The concatenation operator (`//`) appends the suffix `'.flx'` to the Unix file name. Care should be taken with the concatenation operator that each operand be of type `CHARACTER*` (and not, for instance, of type `REAL*8`).

Calls to `POSN` were handled differently in this conversion. For example, in PIODRP, the call

```
      CALL POSN(1,NRD,MFILE)
```

was converted to

```
      integer*4 fopen
      character*8 c8
      character*16 cfile
      .
      .
      .
      lentap=islen(dtape)
      call istring(mfile,c8,8,m)
      cfile=dtape(1:lentap)//'.file'//c8(m:8)
      if (fopen(1,nrd,cfile,0,'VB',32760,32760).ne.0)
     1   call fopenerr(cfile)
```

As in the previous example, the character string dtape must previously be set to a prefix of a Unix file name. The utility function istring was written to assist in this transformation. The subroutine call istring(i,c,n,m) translates the integer i into ASCII characters right aligned in c(1:n). This routine sets m to the integer such that i begins at position m, *i.e.*, c(m:n) contains the translation of i. Essentially, this transformation is constructing a suffix, for example '.file9', the trailing integer of which *indexes* a file in the manner of the call to posn.

The need for a call to fclose is typically indicated by a call to unload in the IBM source code. For example, in the PIODRP routine wrtpha, the call

```
|       CALL UNLOAD(NPHAOT)
```

was converted as

```
|       call fclose(nphaot)
```

One final concern regarding the use of the FTIO library occurs when calls to fread (or fwrite) specify the use of an I/O buffer internal to the FTIO routines. This is requested of the FTIO routines by using a negative unit number to specify a unit. In this conversion, all such specifications were removed. This placed the burden of allocation and management of space for I/O buffering on the Sun source code. An example from the conversion of EDRLST should illustrate this transformation. The IBM source code fragment

```
|      DATA NRD/10/,NEGNRD/-10/,LENCMD/1200/
|      .
|      .
|      .
|      MFILE=2
|      CALL POSN(IOTYP,NRD,MFILE)
|      CALL FREAD(MYADR,NEGNRD,LEN,&890,&860)
|      IF (LEN.NE.LENCMD) GO TO 840
|      CALL UPKCMD(MYADR)
```

was transformed into

```
|      DATA NRD/10/,NEGNRD/-10/,LENCMD/1200/
|      character*1200 bufcmd
|      .
|      .
|      .
|      lentap=islen(dtape)
|      call fopen(1,nrd,dtape(1:lentap)//'.file2',
|     1          0,'VB',32760,32760)
|      CALL FREAD(bufcmd,nrd,LEN,*890,*860)
|      IF (LEN.NE.LENCMD) GO TO 840
|      CALL UPKCMD(bufcmd)
```

In this implementation, of course, upkcmd had to be changed to take a pointer to a block of memory rather than, as before, a pointer to a pointer to a block of memory. More will be said in a later section about the passing of pointers to assembly routines in conversions.

Differences in Hardware

Three differences between the IBM and Sun hardware deserved special attention in this conversion work. These differences are in floating point representations, half-word integer alignments, and character set encoding.

Floating Point Representations

The first hardware difference between the IBM and the Sun to consider is that the two machines have different internal representations of floating point numbers. The primary concern is when reading, on the Sun, floating point data previously written using the IBM representation. When this occurs, the representation needs to be changed to the Sun (IEEE) representation before processing on the Sun. This conversion may be accomplished by the NASA provided `iffe` subroutine.[4] The most opportune place to perform this conversion is immediately after the data has been read (the correctness of this, of course, depends on the usage of the floating point numbers). The following fragment from PIODRP demonstrates this conversion:

```
      COMMON /DATAREC/ MSDATA,MDAYYR,MTMFLG,MDSPR1,SNR,MDSS,MBITRT,
                       MFORMT,MSRTLT,MEXSKD,MDSPR2,MFLAGS,RATTIM,SPNPER,
                       MSPF,ROLPUL,MTM112,BUSVLT,BUSCUR,MTM108,PLTTEM,
                       MDATA(1280)
      .
      .
      .
130   CALL FREAD(MSDATA,NRD,LEN,*150,*133)
      snr=iffe(snr)
      rattim=iffe(rattim)
      spnper=iffe(spnper)
      rolpul=iffe(rolpul)
      busvlt=iffe(busvlt)
      buscur=iffe(buscur)
      plttem=iffe(plttem)
```

In general, bit patterns may be initialized to reals in the IBM format in several ways other than by the `fread`. Two other conditions to watch for are initialization of reals by equivalencing and initialization using `fmove`. Also, care should be taken that `iffe` is not simply applied to every variable of *type* real after initialization, but only to those variables that are used as reals and store reals in the IBM format.

Half-Word Integer Alignments

The second machine difference of concern is the difference in acceptable half-word alignments. On the Sun, half-word integers passed as arguments to subroutines expecting whole-word integers is generally dangerous. As an example, the following code, where `test` is implemented as an assembly routine, is acceptable on the IBM.

```
      INTEGER*2 H(2)
      CALL TEST(H(1))
      CALL TEST(H(2))
      CALL TEST(I)
      STOP
      END
```

[4]Written by Nand Lal (GSFC) and E.S. Panduranga (STX), 1991.

Care should be taken when adapting this code to the Sun. The following solution leads to several problems:

```
INTEGER*2 H(2)
CALL TEST(H(1))
CALL TEST(H(2))
CALL TEST(I)
STOP
END
SUBROUTINE TEST(I)
RETURN
END
```

The first call to `test` passes a surprising value to `test` via `i` instead of `h(1)`. The second call to `test` leads to a violation which crashes the program. For general security on the Sun, subroutine calls which use half-word actual parameters where whole-word arguments are expected should be avoided. Three possible solutions to this lead to one of the following alternatives:

```
INTEGER*2 H(2)
itmp=h(1)
CALL TEST(itmp)
itmp=h(2)
CALL TEST(itmp)
CALL TEST(I)
STOP
END
SUBROUTINE TEST(I)
RETURN
END
```

or

```
INTEGER*2 H(2)
CALL TESTh(H(1))
CALL TESTh(H(2))
CALL TEST(I)
STOP
END
subroutine testh(h)
integer*2 h
return
end
SUBROUTINE TEST(I)
RETURN
END
```

or

```
INTEGER*2 H(2)
CALL TEST(int(H(1)))
CALL TEST(int(H(2)))
CALL TEST(I)
STOP
END
```

```
SUBROUTINE TEST(I)
RETURN
END
function int(h)
integer*2 h
int=h
return
end
```

The second approach was taken in the conversion of the IBM version of `qbit(3,i,j)`, which was called with `i` as either a half- or whole-word integer. Two routines, `qbit` and `qhbit` resulted from this conversion.

The third approach was taken in the conversion of the PIODRP routine `phaout`.

Character Set Encodings

The third hardware difference is that the two machines use different character set encodings (the Sun uses ASCII and the IBM uses EBCDIC). As in the case of floating point representation differences, this problem arises when reading strings which were written in EBCDIC format. A related problem is that of reading numbers in an IBM peculiar *zoned* format. For these purposes, two routines, `cnvebc` and `izone` were written to assist in conversion. The subroutine call `cnvebc(c,l)` converts the first `l` characters of the character string `c` from EBCDIC to ASCII (storing the results in `c`), and the function call `izone(c,l)` converts the first `l` characters of the character string `c` from zoned format to integer (returning the integral result). Essentially, `izone` is used in assembly routine conversion to implement the assembly instructions

```
PACK
CVB
```

The following extraction from the conversion of `upkcmd` demonstrates the use of both `cnvebc` and `izone`:

```
subroutine upkcmd(buffer)
character*1200 buffer
character*8 c
.
.
.
hdycmd=izone(buffer(j:j+2),3)
.
.
.
c=buffer(j+13:j+20)
call cnvebc(c,8)
```

CONVERTING ASSEMBLY ROUTINES

Style and Assembly Routines

In most cases, each IBM assembly language routine was converted to a single routine on the Sun. There were two exceptions to this which merit discussion as general principles of programming style.

The first exception was in the case of `qbit`. On the IBM, `qbit(1,h,j)` would be called as a *subroutine*, and `qbit(3,h,j)` would be evaluated as a *function* which returns a logical value. As this is not good programming practice, `qbit` was converted to both a subroutine, `qhsetbit(h,j)`, and a function, `qhbit(h,j)` (and also `qbit`, as discussed above).

The second exception was in the cases of `pkhet` and `pklet`. As the logic of these two routines is the same, these routines were combined into a single subroutine, `pkhlet`, in the conversion. This was merely a matter of taste, with a consideration for localizing the logic of these routines. More about `pkhlet` will be said in the next subsection.

Passing of Pointers to Assembly Routines

On the IBM, calls to assembly routines pass addresses of the arguments to the assembly routines. For example, the following statement

```
      CALL TEST(H(1))
```

where `test` is an assembly routine and `h` an array, would pass the address of `h(1)` to `test`, thereby giving the routine `test` access to all of `h`. This trick was used at several places in the IBM source code. Conversion of the assembly routine to C should allow the call to go as is; however, conversion of the assembly routine to FORTRAN requires some doctoring of the subroutine call. As an example, the following fragment is from the PIODRP routine `phaout`:

```
      IMPLICIT INTEGER*2 (H)
      DIMENSION HD(12),HPNA(120,6)
      .
      .
      .
      CALL PKLET(HD(3),HPNA(118,6),&72)
```

As discussed above, `pkhet` and `pklet` were translated into `pkhlet`. The above call gives `pklet` the addresses of `hd(3)` and `hpna(118,6)`, and `pklet` accesses components of the arrays `hd` and `hpna` beginning at those addresses. The idea of the transformation applied is to create character strings equivalenced to `hd` and `hpna`, and then pass the proper substrings to `pklet`. If `chd` and `chpna` were character strings of appropriate sizes and were equivalenced to `hd` and `hpna`, then the following subroutine call

```
      CALL PKLET(cHD(5:),cHPNA(2869:),*72)
```

would be an appropriate conversion. As discussed in the above section, `pkhet` and `pklet` were converted into a single routine, `pkhlet`. This routine takes an additional first argument of `0` to perform the `pkhet` function, and `1` to perform the `pklet` function. The final transformation of the above fragment results in the following:

```
      IMPLICIT INTEGER*2 (H)
      DIMENSION HD(12),HPNA(120,6)
      character*24 chd
      character*1440 chpna
      equivalence (hd(1),chd),(hpna(1,1),chpna)
      .
      .
      .
      CALL PKhLET(1,cHD(nhd(3):),cHPNA(nhpna(118,6):),*72)
```

The utility functions nhd, nhpna, nphnb, nhpga, and nhpgb were written to compute the proper indexes into equivalenced character strings used by phaout.

Converted Assembly Routines

The following assembly routines have been converted to the Sun:

2 contim	3 logdec	1 pklet	2 upklog
1 declog	2 mstot	2,3 qbit	2 upkseq
1 getput	2 mvzero	2 upkcmd	2 upkspn
2 hgroup	1 phasrt	2 upkflg	2 upktim
1 iget	1 pkhet	2 upkfmt	

Three techniques were employed in the conversion of a routine. The first technique was the series of transformations from one language to another described in the above section on goals and principles. The second technique was that of reading the initial specifications for the routine, writing a routine from scratch based on those specifications, and then testing the functionality of the two routines for equivalence using the differences as a refinement of the initial (typically ambiguous) specifications. This may be thought of as a special case of the first technique where the initial transformations are given. The third technique was to treat the routine as a black box, determine the functionality of the routine by establishing relationships between inputs and outputs, and then using this functionality as the specification of the routine. Essentially this technique derives the specifications of the routine without regard to the source code or initial specifications. This technique was employed in the conversion of logdec.

CHANGES TO THE OPERATIONAL ASPECTS OF THE TOOLS

Global changes in the way the converted system will ultimately function on the Sun directly influenced the conversion effort. The most significant change affecting the conversion work was the termination of the existing catalog system. The approach taken was to simulate the reading of the catalog that exists on the IBM. An example of this occurred in the conversion of the FLXDBG routine dbgdrs. The following code from this routine was reads catalog data from the tape catalog:

```
      COMMON /DRSCAT/ HIDSPC,HSPAR,HPHATP,HRATTP,DPHATP(100),
     1 MSPHAS(100),MSPHAE(100),HDPHAS(100),HDPHAE(100),HPHAFT(100),
     2 DRATTP(100),MSRATS(100),MSRATE(100),HDRATS(100),HDRATE(100),
      .
      .
      .
200   READ(20,210) NCATP,HID
210   FORMAT(I1,A1)
      NCUNP=NCATP+20
      CALL FREAD(HIDSPC,NCUNP,LEN,&520,&610)
      CALL UNLOAD(NCUNP)
      IF (HID.NE.HIDSPC) GO TO 500
      RETURN
```

This was converted as

```
      common /influx/ dsat,asat,...
      common /source/ dtape
      COMMON /DRSCAT/ HIDSPC,HSPAR,HPHATP,HRATTP,DPHATP(100),
     1 MSPHAS(100),MSPHAE(100),HDPHAS(100),HDPHAE(100),HPHAFT(100),
     2 DRATTP(100),MSRATS(100),MSRATE(100),HDRATS(100),HDRATE(100),
      .
      .
      .
      data asatf/'F'/,asatg/'G'/,hsatf/'F'/,hsatg/'G'/
      NCUNP=NCATP+20
      if (asat.eq.asatf) then
         hidspc=hsatf
      else if (asat.eq.asatg) then
         hidspc=hsatg
      else
         write(6,211)
         call abend(-1)
211      format(1x,'Character 9 of SATID must be either F or G')
      endif
      hid=hidspc
      hphatp=1
      dphatp(1)=dtape
      hdphae(1)=32000
      hrattp=1
      drattp(1)=dtape
      hdrate(1)=32000
      IF (HID.NE.HIDSPC) GO TO 500
      RETURN
```

The basis for this approach is the maintenance of assertions essential to the correctness of the program, and the motivation for this approach is the concern for a quick conversion. In the overall scheme, once the program is working with these transformations, then more comprehensive changes could be considered. The objective, again, was to break the dependency on the IBM as soon as possible.

CONVERTING LARGER SYSTEMS

After the conversion of assembly routines and standard transformations of routines, there likely remains a great deal of work to be expended in putting all the pieces together to build a functioning system. In the Pioneer production conversion effort, a combination of two approaches was employed.

In the first approach, a bottom-up approach, the source code was first analyzed for dependencies between modules. The dependencies were represented by a directed graph (acyclic in this case), and the sinks (*i.e.,* those modules which were independent of other modules) were identified. The approach was to begin with the modules which were sinks and work up the hierarchy to those modules which were sources (*i.e.,* those upon which no other modules were dependent). Essentially, this approach attempts to convert the software in the same order that it would have initially been built.

The second approach, a top-down approach, attempts to convert the software in the order in which modules are called and statements executed, as if tracing the execution of the program. This method proved particularly useful in setting up conditions for invoking subroutines being converted in the order chosen in the first approach above.

State of the Conversion

At the time of the preparation of this document, the Pioneer production software has been tested against the IBM on Pioneer-F data in formats A and A/D. Additional and more extensive testing should be performed against the IBM. More specifically, testing of Pioneer-G data and Pioneer-F data in B formats should be performed.

The format of the testing to date has been the comparison of outputs from the listing programs PFRTPL, PRNPHA, and FLXLST run on both the Sun and the IBM. Inputs to these listing programs were the output files from PIODRP and FLXDBG. Comparison of the resulting text files was made with the Unix `diff -h` command.

No catalog system has been specified or implemented for the data at this time, either. This work will possibly involve major changes to the software, resulting in the need for further testing. The initial Sun conversion, once tested and proven, can serve as the basis for these future tests.