Chapter 1

Introduction

1.1 Scope

This document is intended for programmers who are familiar with general programming practices and FORTRAN in particular and who are familiar with the common techniques for manipulating astronomical data. This manual is intended to be of use to casual as well as serious programmers wishing to program using the AIPS system. *Going AIPS* is not intended to be an exhaustive description of the functions and subroutines available in AIPS, but rather to illustrate general techniques.

1.2 Hey You, Read This

This manual is designed for a wide variety of users, ranging from those wishing to add 1 line of code to an existing task to the poor soul who has to assume the care and feeding of AIPS in the case all the current AIPS programmers are hit by a truck. While the weight of this manual would tend to bring on attacks of massive depression or homicidal mania in the lighter users from the above mentioned range, it should be noted that, for many purposes, only a small fraction of the material in this manual is necessary in order to program in the AIPS system. The following table suggests courses of action for various situations.

- "I want to get my data into AIPS."

  There are a number of skeleton tasks which make this relatively straightforward — frequently requiring several hours of effort. See the chapter on the skeleton tasks and ignore the rest of this manual unless you run into problems.

- "I just want to do something simple to my data."

  See the chapter on skeleton tasks. There are two tasks, FUDGE and TAFFY, which read uv data or an image, pass the data to a user-provided subroutine and write what comes back into a new file. All of the messy stuff is already taken care of.

- "I have this idea."

  This requires a bit more understanding about how AIPS works. Read the rest of this chapter, the chapter on the skeleton tasks, the chapter on tasks, and the chapter on disk I/O. Depending on the application, several other chapters may be relevant. Then find an existing task that is closest to your need and start from there. For a great many purposes the skeleton tasks are a good place to start. There is also a chapter describing various high-level utility routines such as making images from uv data or subtracting model values from uv data.

- "I have lots of ideas."

  Find a comfortable chair, open a six pack of beer and start reading.

- "We just bought the Whizbang 8000 computer and want to run AIPS on it."

  Read all of this manual especially chapters 10 and 15 and Appendix A, then give us a call.

- "Why didn't you %#&(*&! see that #&*@!%̂ truck."

  Read it all, then write the parts left out. Lots of luck.

1.3 Philosophy

The NRAO Astronomical Image Processing System (AIPS) is designed to give the astronomer an integrated system of flexible tools with which to manipulate a wide variety of astronomical data. To be of maximum benefit to the general astronomical community and to increase the useful lifetime of the software, the AIPS system has gone to great lengths to isolate the effects of the particular computer and installation on which it is run. Needless to say, this portability requirement makes the programmer's life more difficult.

The routines which depend on the host machine or operating system are denoted by using a "Z" as the first character of the name; these are referred to as the "Z routines". No other "standard" routines should depend on the host machine or operating system to work properly. Routines which depend on the particular television display device are denoted with names beginning with a "Y"; these are the "Y routines". Routines which depend on the computing hardware (e.g., array processors, vector processors, or lack thereof) have names beginning with a "Q".

It has been argued that it is not worth the additional effort to isolate the machine dependencies. We are all aware of usable packages that have died because they were strongly tied to a particular computer. VAXes are currently losing their position of dominance in the astronomical computing community and those with a sufficiently long memory will recall that IBM 360s and 370s and CDC Cybers had a similar stranglehold during the 60s and early 70s. By not tying ourselves to a particular computer or even vendor, we have the freedom to buy hardware from the vendor who offers the most cost-effective models. This strategy should allow the AIPS system to last longer than previous systems, so we can spend more time investigating new algorithms and less time patching or recoding old programs every time we change computer.

In addition to isolating machine dependencies, we advocate modular program structure. By this we mean that the main program should be relatively short and should basically call routines each of which has a well defined and limited function. Modular coding is especially important for machines on which most programs must be overlaid (a dying species), but it also makes the code easier to debug, easier to maintain, and very importantly, easier from which to steal pieces. Routines which may be of use in other applications should be coded in as general a form as possible and placed in the appropriate AIPS subroutine library. This may take longer in the short run, but should pay off in the long run.

Another philosophical feature of AIPS is that the programs should run as quickly as possible without making the code too difficult to maintain. This is frequently a matter of judgment, but, in general, tricks and excessive cleverness should be avoided.

Since many of the most expensive AIPS tasks are I/O limited, the AIPS I/O system has been designed for maximum performance. In general, this means that I/O is done in a double-buffered mode, in as large blocks as possible, with fixed logical record size and programs work directly out of the I/O buffers. This makes many of the features of the I/O system, which are normally hidden from the programmer, much more obvious and allows the I/O to run as fast as the computer can manage.

The AIPS philosophy has always been that it should always be possible to determine what has been done to a data set. For this purpose, every permanent cataloged data file has an associated history file in which a permanent record is kept of the processing done to the data in that file. It is the responsibility of the programmer to insure the integrity of the history. In addition to the history files, most communications between the user and AIPS or tasks are logged in a file which can be printed.

1.4 An Overview of the AIPS System

The AIPS system consists of several distinct parts. First and most obvious to users is the program called AIPS. This program, based around the People Oriented Parsing System (POPS), interacts with the user, performs many of the display functions, does some manipulation of data and initiates other programs which run asynchronously from AIPS. Functions built into AIPS are called verbs, the asynchronous programs are called tasks, and both are controlled by the values of parameters in the POPS processor known as adverbs. A third type of program in the AIPS system is the stand-alone utility program which is mostly of interest to the AIPS system manager.

1.4.1 Tasks

Communication between the AIPS program and the tasks it spawns is fairly limited. When a task is initiated from AIPS, an external file is read which specifies the number and order of adverbs whose values are sent to the task. These values, along with some "hidden" values, are written into a disk (TD) file. AIPS then initiates the requested task and begins looping, waiting for the task to either disappear or put a return code into the TD file. The task reads the TD file and depending on the value of a logical "hidden" adverb (DOWAIT in AIPS and RQUICK in the task) may immediately restart AIPS by returning the return code. The task then does the requested operation and before stopping, sends AIPS the return code if this was not done previously.

AIPS may communicate with a task after it has started running via the task communication (TC) file. A list of adverbs which are to be sent to the task is defined in the inputs file; in addition, other instructions such as "quit" may be sent. The task must read the TC file at relevant points. It is the responsibility of the programmer to check the TC file and take appropriate actions.

Tasks are used for operations which either require much computer memory or CPU time or both, whereas verbs are used for operations which take no longer than a few seconds to finish. Since the tasks run asynchronously from AIPS, the user may do other things while one or more tasks are running. Since there is a minimal interaction between AIPS and tasks, programming tasks is much simpler than programming verbs; AIPS does not need to be modified to install a new task. Tasks may communicate directly with the user.

1.4.2 Verbs

Verbs are the functions built into the AIPS program itself. Many of these involve the display of images and most of the interactive features of the AIPS system. POPS is a programming language itself, and complicated combinations of tasks and verbs may be assembled into POPS procedures. Verbs, but not tasks, may change the value of POPS adverbs.

The AIPS program is very modular and most verbs are implemented via a branch table contained in an external file. Most of the verbs are called from subroutines with names like AU1, AU2, AU5C etc. A table read from an external text file determines the subroutine and a function number for each function. The values of adverbs are contained in a common.

1.4.3 Data Files

Data is kept in files which are cataloged in AIPS. At present we have two kinds of data (more are possible): images and uv data. The internal structure is much like that of a FITS format tape. Associated with each main data file may be up to 20 types of auxiliary information files with up to 255 versions of each type. The basic information about the main data file and the existence of the auxiliary files (called extension files) is kept in a catalog file. Bookkeeping and other information is kept in the first record of most of the extension files. One example of the extension file is the HIstory file in which a record of the processing of the data is automatically logged by the AIPS tasks.

1.4.4 I/O

The AIPS system has three basic types of files and three types of I/O to access them. The main data files which are assumed to contain the bulk of the data are accessed in a double buffered mode with large blocks being transferred. The extension files are read by single buffered transfers of 256 integers. Both types are intrinsically random access; however, in practice the main data file access is sequential, but the extension file access is frequently random. For the main data file, I/O tasks usually work directly from the I/O buffer. The third type of file is the text file. More details about the I/O routines can be found in the chapter on I/O.

1.5 Style

Since AIPS is a rather large package maintained by numerous people it is important that all of the software be written in a consistent style. The following sections describe the style in which AIPS software is to be written.

1.5.1 Precursor Comments

Precursor comments are the principal form of detailed programmer documentation in the AIPS system. These are comments placed immediately following the PROGRAM, SUBROUTINE, or FUNCTION statement which explain the purpose and methods of the routine, the input and output arguments, any use of variables in commons, and any special coding techniques or limitations in the transportability of the routine. Precursor comments do not need to be verbose, but they must explain most things which a programmer must know about calling the routine. Routines must have acceptable prologue comments before they will be accepted into the AIPS system.

The precursor section should begin with two coded comment lines which give the use of the routine and placing it in one of a number of categories. The first of these is a one line description of the function of the routine; this line begins with a comment delimiter followed by a "!". The second line lists the categories in which the routine fits; this line begins with a comment delimiter followed by a "#". These two lines allow the automatic generation of software documentation.

Following the coded routine description lines is a user agreement notice which is intented to discourage anyone from selling the AIPS software. Precursor comments describing the functionality of a routine should be indented three columns except for the indentation of new paragraphs. Sections describing input or output call arguments or major common variables should be set off by an "Inputs:" or "Outputs:" etc. line also indented three columns. Descriptions of individual variables should be indented 6 columns and consist of three parts: 1) variable name, 2) type and dimensionality and 3) function, units etc. These parts should be arrainged into columns. The declarations of the call arguments should be separated from the declarations of the local variables by a single (mostly blank) comment line.

As a simple example, consider:

```
      SUBROUTINE COPY (N, KFROM, KTO)
C-----------------------------------------------------------------------
C! copies integer words from one array to another
C# Utility
C   This software is the subject of a User agreement and is confidential
C   in nature. It shall not be sold or otherwise made available or
C   disclosed to third parties.
C-----------------------------------------------------------------------
C   COPY transfers N integer words from KFROM to KTO
C   Inputs:
C      N      I        number of words to be copied
C      KFROM  I(N)     input array
C   Outputs:
C      KTO    I(N)     output array
C-----------------------------------------------------------------------
      INTEGER   N, KFROM(*), KTO(*)
C
      INTEGER   I
C-----------------------------------------------------------------------
      IF (N.LE.0) GO TO 999
```

```
            DO 10 I = 1,N
               KTO(I) = KFROM(I)
 10            CONTINUE
C
 999  RETURN
      END
```

1.5.2 Body Comments

"Body" comments are placed at strategic locations throughout the body of the code. They act as sign posts to alert the reader to each logical block of code and also to clarify any difficult portions. Ideal places for body comments are prior to DO loops and IF clauses. Body comments within a routine must all begin in the same column and that column should be near column 41. Body comments (and precursor comments) should be typed in lower case letters. This helps to separate visually the comments from the program text (which must be all in upper case!!!).

1.5.3 Indentation

Another powerful tool to illustrate to the reader the logical structure of a routine is indentation. By indenting statements to indicate that they belong together, one can enhance greatly the readability of one's programs. Each step of indentation shall be three (3) spaces, beginning in column 7. Numbered CONTINUE statements should be employed to enhance the indentation pattern. DO loops and IF clauses should be indented. The final CONTINUE and END IF statements should be indented the same as the bulk of the loop or IF block. ELSE or ELSE IF statements should be indented the same as the corresponding IF statement. As an example, consider:

```
C                                       Multiply by transform matrix
      DO 20 I = 1,3
         VEC(I) = 0.0
         DO 10 J = 1,3
            VEC(I) = VEC(I) + TMATX(I,J)*VECO(J)
 10         CONTINUE
 20      CONTINUE
C                                       Unit vector to polar
C                                       Case at pole
      IF ((X.EQ.0.0) .AND. (Y.EQ.0.0)) THEN
         ALPHA = 0.0
         DELTA = 0.0
      ELSE
         ALPHA = ATAN (X, Y)
         DELTA = SQRT (X*X + Y*Y)
         END IF
      PDIST = ATAN2 (Z, DELTA)
C                                       Swap to increasing order
      IF (A.GE.B) THEN
         C = A
         A = B
         B = C
         END IF
      Z = Z ** (B-A)
```

1.5.4 CONTINUE Statements

All DO loops end with CONTINUE statements rather than some executable statement. This enhances legibility as well as preventing compilation errors on those statements which are not allowed, by some compilers, to be the last statement in a DO loop.

1.5.5 Statement Numbers

The use of GO TO statements is the cause of most logic errors in programming. Use of IF-THEN-ELSE constructions can frequently simplify the logic of a routine. Statement numbers must increase through the routine and should be integer multiples of 5 or 10. They should not exceed 999. Format numbers should have 4 digits with the low order 3 giving the nearest preceding statement number to the first statement using that format. All statement numbers are left justified beginning in column 2.

Statement numbers can help to clarify the logical structure of a routine. Let us consider the common example of a routine which begins with some setup operations (e.g., file opening), then does operation set A or B or C or D, and then does some close down operations (e.g., file closing) before returning. Where possible, such a routine should use statement numbers 5–95 for the setup, 100–195 for set A, 200–295 for set B, 300–395 for set C, 400–495 for set D, and 900–995 for the close down. All FORTRAN routines should end with a RETURN or STOP (main programs only) statment labeled 999.

1.5.6 Blanks

Blank spaces can improve the readability of the routine as can parentheses. Blanks should surround equals signs and separate multiple word statements. Parentheses are a great help in compound logical expressions. For example,

```
A = B
DO 10 I = 1,10
GO TO 999
CALL KPACK (IX, IY)
IF ((A.GT.B) .AND. (C.LE.D)) THEN
```

1.5.7 Modular Code

Modularity in program design is a very important asset for many reasons. Complicated tasks become clearer, to coder and reader alike, when constructed from a logical sequence of smaller operations performed by subroutine call. Such well-ordered tasks are far easier to design, to understand, and to make work correctly than vast monolithic single programs. Furthermore, the small operation subroutines will often turn out to be fairly general and useful to many other tasks as well. Programmers will have to remember that their tasks will have to run not only in the "unlimited" address space of 32-bit virtual computers, but also in the very limited address space of older computers. Although many modern computers have large ($\geq$ Mbyte) memories, these memories are often divided amongst many users. Also many older AIPS computers have limited memories. Therefore, programmers must remember that excessive page faulting is extremely expensive on most virtual memory computers.

1.5.8 Portability

The code of AIPS is intended to achieve a very high degree of portability between computers. The machine independent portion of the AIPS software must strictly conform, after preprocessing, to FORTRAN 77 rules. Vendor extensions to the language are not allowed. Extensions to FORTRAN allowed by the FORTRAN preprocessor are described later in the section on FORTRAN. In particular CHARACTER and numeric data are not to be equivalenced or mixed in a common.

All of the things mentioned in this chapter should be used in moderation. One can bury good code in a plethora of inane comments. One can inundate statements with parentheses or spread them out with blanks until they are no longer legible. Vastly elaborate indentation and numbering schemes can confuse rather than aid the reader. The creation of large numbers of very short, special purpose subroutines will overburden linkage editors and AIPS's bookkeeping schemes. (In this regard, AIPS already contains a wide range of useful utility subroutines. Programmers should check to see if a function is already available before creating additional subroutines.) Basically, programmers should use good common sense in applying the standards described in this chapter.

1.6 Language

The magnitude of the AIPS project and the desire to achieve portability of the software require a high degree of standardization in the programming language and style. One must code in a language which can be compiled on all machines. One must follow strict rules in statement ordering and location so that simple preprocessors may, when necessary, locate and modify the standard code. Everyone must type code in the same way so that all programmers will be able to read it with as little effort and confusion as possible. All experienced programmers develop a personal typing style which they prefer. To them, the rules given in this chapter may seem arbitrary, capricious, and unworkable. Nonetheless, they are the rules to be followed when coding for the AIPS system. Routines which do not meet these standards will not be accepted. This project is too important and too large to allow compromise at this level. Also, we have found these rules to be fairly comfortable — after we got used to them.

1.6.1 FORTRAN

The programming language will be ANSI standard FORTRAN 77, except for the addition of INCLUDE, LOCAL INCLUDEs, and HOLLERITH declarations and the use of a minimum number of local assembly language (or C) Z routines when absolutely required. The extensions to FORTRAN 77 will be translated by the preprocessor to standard FORTRAN 77. The preprocessor will include the text of INCLUDE files, allowing the definition of "local INCLUDES" in the file and translate HOLLERITH declarations to a numeric data type.

Hollerith data is characters coded into a numeric type variable. The use of Hollerith data is required in some circumstances by the prohibition in FORTRAN of mixing CHARACTER and numeric data in a common or of EQUIVALENCEing them in any way. Since the length of a character string in terms of numeric data types is not defined character type data and numeric data may not be mixed in fixed length records.

Due to these restrictions on the use of CHARACTERs in FORTRAN, AIPS uses data type HOLLERITH in a limited set of circumstances, most notably in file catalog headers which are data structures containing numeric and character data. All translation between HOLLERITH and CHARACTER data types is done in the routines CHR2H and H2CHR which are described in Chapter 3. The only operation allowed for HOLLERITH type data is the assignment to another HOLLERITH variable. HOLLERITH variables must NEVER appear in DATA or WRITE statments.

AIPS FORTRAN requires that all variables be declared. This requirement, when enforced by the compiler, is a valuable tool for finding typos and related bugs.

A review of the entire language is inappropriate here, but programmers are urged to reread a basic reference. (Do not read your local VAX FORTRAN manual. Use a fundamental reference such as IBM's FORTRAN Language manual.) In particular, programmers are reminded that the names of commons, variables, functions, and subroutines must begin with a letter and contain no more than six (6) characters. In AIPS, program names may have no more than five characters because of the need to append the value of NPOPS. Comments are introduced by placing the capital letter C in column 1 of the card. No in-line comments are allowed. Continuation statements are formed by placing a non-blank character in column 6 of the card. In AIPS, this character shall be an asterisk (*). There may be no more than 19 continuations of a single statement. Only card columns 1–72 are used, even in comments. Executable statements at the first level of indentation begin in column 7. TAB characters must not be left in the code after it is typed and edited. The three non-standard statements have the forms:

1. INCLUDE 'INCS:<name> '

where INCLUDE begins in column 7, the first single quote is in column 15, the <name> is a left justified character string. and the second single quote follows <name> with no blanks. The conventions for <name> will be described later. The statement causes the file called <name> to be inserted in the routine in place of the INCLUDE statement. The INCS: indicates the standard include area or search path and should be omitted for "files" given by LOCAL INCLUDEs. Only a single level in INCLUDE is allowed.

2. LOCAL INCLUDE '<name> '

   where LOCAL starts in column 1 tells the preprocessor that the text following, up to the next "LOCAL END" also starting in column 1, is to be included when a "INCLUDE '<name> ' " line is encountered. LOCAL INCLUDEs are normally defined at the beginning of the file containing a task and should only contain text relevant to that task, e.g. defining it's internal commons.

3. HOLLERITH <list>

   where <list> gives the list of variables to be declared as type HOLLERITH. The AIPS usage of HOLLERITH is that 4 characters may be stored in each element.

1.6.2 Statement Order

Statements must be ordered as follows. The PROGRAM, FUNCTION, or SUBROUTINE statement must occupy the first line and must begin in column 7. Then come the precursor comments, the declaration statements, the body of the program, the format statements, and the END statement. Each of these segments will be separated by a comment delimiter line (i.e., C followed by 71 minus signs). The last delimiters are omitted if there are no FORMATs. The last line of the body of the routine must have the statement number 999 and be a STOP (for programs) or RETURN (for functions and subroutines) statement. There must be no other STOP or RETURN statement in the routine.

Many computer systems allow declaration statements to occur in almost any order. However, FORTRAN and some of the simpler compilers do not. Therefore, in AIPS, we will use the following order:

1. Data type and dimension statements: HOLLERITH, INTEGER, LOGICAL, REAL, DOUBLE PRECISION and COMPLEX in any order. We prohibit DIMENSION and data types not allowed by FORTRAN 77 (excluding HOLLERITH). and any use of these statements for data initialization. Note: the use of COMPLEX arithmetic is discouraged as many compliers do not correctly compile statements involving complex arithmetic. PARAMETER statements should be included with (usually before) the declaration statements. Declaration, EQUIVALENCE and COMMON statments may be mixed.

   We prohibit use of the COMMON statement to give the types and dimensions of variables. Use of blank common must be reserved for cases where dynamic memory allocation is needed and the blank common can be changed in size.

2. Data initialization statements: DATA. We prohibit the use of DATA statements to initialize variables in commons (as do the FORTRAN standards and many compilers). The use of octal and hexadecimal numbers in data statements is forbidden.

3. Function definitions.

1.6.3 INCLUDEs

INCLUDE statements are used in AIPS primarily to provide a fixed and uniform set of declarations for commons and data structures. The naming conventions for such INCLUDEs is 'INCS:accc.INC', where INCS: is a logical directory name (which must be dealt with by the preprocessor), 'a' is P, D, or V for PARAMETER INCLUDEs (include files defining PARAMETERS), Declaration/EQUIVALENCE/COMMON includes and includes containing DATA statments. These INCLUDEs must be named in this order.

```
      INCLUDE 'INCS:DBWT.INC'
```

causes the text:

```
C                                                     Include DBWT.
      INTEGER   BWTNUM, BWTLUN, BWTIND, BWTREC, BWTDAT(256)
      LOGICAL   WASERR
      CHARACTER BWTNAM*48
      COMMON /BWTCHC/ BWTNAM
      COMMON /BWTCH/ BWTDAT, BWTNUM, BWTLUN, BWTIND, BWTREC, WASERR
C                                                      End DBWT.
```

to be inserted. Note that CHARACTER variables are in a separate common from numeric variables.

1.6.4 Dimension Declaration

The declaration of the dimensionality of arrays should be done as accurately as possible. When arrays are passed as call arguments and the leading dimension is not passed, declare tha array “(*)” and never as “(1)”. Arrays which are declared and equivalenced to other variables should be declared as accurately as possible.

1.6.5 SAVE Statements

If the value of a local variable in a subroutine or function is to be preserved between calls, it should be mentioned in a SAVE statement. Some but, not all, compilers do this automatically but it is not required by the definition of FORTRAN.

1.7 Documentation

Proper documentation for both users and programmers is vital to the success of any software system. In the AIPS system, this documentation is primarily the responsibility of the programmer. In the following sections the various categories of AIPS documentation are discussed.

1.7.1 User Documentation

HELP files

The primary source of user documentation is the HELP file. This information is available to the user on-line from the AIPS program. There are several types of help files: (1) task help files, (2) general help files, and (3) adverb help files. The general help files aid the user in finding the name of the task or verbs for a given operation. These entries consist of the name and a one line description of a task or verb. New tasks should be entered into the appropriate general help files. Task help files are the primary user documentation for a task or verb.

There are three parts of the task HELP file separated by a line of 64 -'s. Details about the format of the HELP file are found in the chapter on tasks.

1. INPUTS

   The INPUTS section of the help file is *required* for any task to run. AIPS uses this section to determine the number and order of adverbs to be sent to the task and can check on limits on the values. The INPUTS section also contains a short description of the use of the task and of each of the adverbs. A listing of the INPUTS section of the help file is displayed on the user's terminal showing the current values of the named adverbs when the user types “INPUT” to AIPS. The INPUTS section is also used to specify any adverbs which may be send to the task during its execution through the TC file.

2. HELP

   The HELP section of the help file gives a more detailed description of the function of the task and a more complete description of the meaning of each of the adverbs than the INPUTS section. This section should also explain the default values of the adverbs. The HELP section of the HELP file is listed on the users terminal when the user types “HELP name”.

3. EXPLAIN

   The EXPLAIN section of the help file should describe the techniques for properly using the task; hints about reasonable values of the adverbs can be given here. A discussion of the interaction of the given task with other tasks is also appropriate. It is best if someone other than the programmer writes the EXPLAIN section of the help file. The HELP and EXPLAIN sections of the help file are written on the line printer when the user types "EXPLAIN name" to AIPS.

AIPS *Cookbook*

The AIPS *Cookbook* is the main User documentation for AIPS. However, many users are unaware of the existence of any feature in AIPS not advertised in the *Cookbook* and unfortunately, the *Cookbook* only covers the most elementary portions of the AIPS system.

1.7.2 Programmer Documentation

Precursor Comments

The most fundamental source of detailed programmer documentation in the AIPS system are comments in the source code, especially the precursor comments. The precursor comments for all routines should describe the use of the routine as well as the meaning, units, etc., of all call arguments. Many of the detailed descriptions of call sequences in this manual are essentially the precursor comments of the routines.

Shopping Lists

The precursor comments of routines contain one line descriptions of the routines. These are used to generate the shopping lists found in Appendix B.

CHANGE.DOC

Once source code, text files, etc. are entered into the AIPS libraries all changes should be documented in the CHANGE.DOC file. Installations outside of the main AIPS programming group are encouraged to adopt this system. The CHANGE.DOC file contains entries giving the date, name of the routine, and the name of the person making the change, with a short description of the changes. If a bug is being corrected, its symptoms should be described.

The Checkout System

The AIPS group has instituted a check-out system for the text files in the master version of the AIPS system (including CHANGE.DOC). The purpose of this check out system is to prevent different programmers from destroying each others changes to code by trying to work on the same routines at the same time. There are occasionally changes made in AIPS which require changes in most or all tasks; frequently the original programmer of a task will be unaware of these changes. For these reasons, modifications or additions to the the master version of AIPS should (are required to):

1. Check out the relevant files. A detailed description of the current check-out routines may be found in DOCTXT:CHKOUT.RNO.

2. Modify the files.

3. Check the files back in.

4. Document the changes in CHANGE.DOC (which must itself be checked out).