Hipparcos and Tycho Data structures and Load routines in C

1. Introduction

The src directory contains code for loading records from every data file (.dat) supplied on the Hipparcos/Tycho ASCII CD-ROM set. Code is also provided which loads and searches all of the associated index (.idx) files with the exception of charts.idx and curves.idx.

This code needs to be compiled on the host system (see Section 6). It is intended that these routines be modified by users to suit their specific needs.

The index routines use these load routines and data structures, and simple programs are provided to use the index files to find records. These are described below.

A simple Makefile is also provided to compile and link these executables.

All of the C header and test routines have identical structure and functions, and they differ only in name (with the exception of the utils file). Hence the structures and routines are described in general, but the find and search routines (which vary from file to file) are described in full.

This document is split into the following parts:

- Section 1: this introduction.
- Section 2: describes the search programs (files prefixed with 's').
- Section 3: describes the data structures which exist for each datafile.
- Section 4: describes the index routines (files prefixed with 'i').
- Section 5: describes the utils files (functions and structures used by all programs).
- Section 6: describes the compilation process.
- Section 7: is a note on unix utilities.
- Section 8: gives some general pointers and information.

In this document keywords and names which appear in the code appear in `courier` font.

When browsing the PDF version of this document with `acroread` all cross references are Hyperlinks. For example clicking on section names in the above list will cause `acroread` to jump to the appropriate page.

2. Search programs (sdatafile.c)

Search programs have been written to demonstrate the use of the index (see Section 4) and access routines (see Section 3). Anyone wishing to write their own code could use one of these routines as a starting point. To get started quickly without finding out how these routines work see Section 8.2.

The search programs have the name of the index file with the letter 's' prepended and all underscores removed. In the case of `hip_ep_e`, which shares the index hip_ep.idx with `hip_ep`, the search routine has been called `shipepe`.

No '.h' files exist for these routines, only '.c' files, but when a `make` (see Section 6) is performed a binary of the same name will be created. Thus we have the following:

```
shipmain
shipi
shipj
shipep
shipepe
shipva
shipdm
shipdmc
stycmain
stycep
ssolar
```

Additionally search routines based on the datafile name have been provided. These may be seen as redundant routines, since any data they retrieve can also be retrieved using the main search routines listed above. This applies to the following:

```
shipdmg
shipdmo
shipdmv
shipdmx
shiprgc
shipva1
shipva2
sdmsao
shpauth
shdnotes
shgnotes
shpnotes
```

2.1 Outline of the search programs

The search programs are all constructed in a similar manner. As an example see the listing for shipmain.c (page 4). The first code to be seen in any of the search programs is the list of 'include' files, all routines include the standard libraries and utils.h files:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "utils.h"
```

Additionally the data structures and indices to be used are loaded, e.g for shipmain:

```
#include "hip_main.h"
#include "ihipmain.h"
```

Next a main routine is defined.

The search program defines one variable of the type of data to be looked at e.g. for hip_main 'hip_main anentry;' is defined. The variable headPrinted is used later to determine if a header has already been printed.

If no arguments are given on the command line the program issues an error message; otherwise it enters a for loop from 1 to argc.

This scans the command line for options and identifiers. There are several command line options which may be passed to the program to change its mode of operation. These are all checked for by the function checkIfCommandLineFlag (see Section 5.3.18). This sets some variables which have been passed to it, namely i cols decode multiRecs.

Next it looks for identifier values, for each identifier on the command line it attempts to retrieve that record (usually using search_) and print it.

If the cols variables has been set to 1 then the data is printed using the tabular print method print_'data_type'_cols (see Section 3.2.4.2). Otherwise the normal print method is called e.g. print_'data_type' (see Section 3.2.4.1). In general a header is printed (see Section 3.2.4.3). However for the search routines which may return records of different types this is not done (i.e. shipdm, shipva and ssolar).

The default print option prints each attribute plus a description for the main record. Any sub-records are printed in tabular mode. Furthermore if -t is specified the main record is printed in tabular mode which, by default, does not print the sub-records. The options -ts need to be specified to get sub-records as well.

Valid options are (see also Section 5.3.18)

```
h - print this message
a - apply c, b, s together
b - bit field decoding (HEPA and TEPA)
c - correlation coefficient extraction (applicable to DMSA/C, G, O, V)
s - sub-record print
t - tabular print
q - supress header from -t
```

Some of the programs support +n to give the next n records after the requested record. If the specified record does not exist then no records will be retrieved. The default operation prints the retrieved record(s) in verbose format (see Section 3.2.4.1).

Here is an example listing of shipmain.c:

```
/*  Hipparcos ASCII CD-ROM load and search routines Release 1.1 June 1997
    William O'Mullane
    Astrophysics Division, ESTEC, Noordwijk, The Netherlands.
    See the readme.pdf file for more information */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "utils.h"
#include "hip_main.h"
#include "ihipmain.h"

main (int argc,char **argv)
{
   hip_main anentry ;
   int headPrinted=0;

   if (argc == 1)
   {
        fprintf (stderr,"You must pass a key value on the command line \n");
   }
   else /* some numbers on the command line */
   {
     int i,decode=ASRAW,cols=0,quite=0;
     long multiRecs=0,recNo;
     FILE* dataFile;
     for (i=1; i<argc ; i++)
     {
        INT rec;

        checkIfCommandLineFlag(argv,argc,&i,&cols,&decode,&quite,&multiRecs);
        if (cols==1 && headPrinted==0 && quite==0)
        {
           print_hip_main_header ();
           headPrinted=1;
        }

     strAsINT( argv[i], &rec);
     if ((dataFile = search_hip_main (&rec,&anentry)) != 0)
     {
         for (recNo=0; recNo<=multiRecs; recNo++)
               {
                  if (cols)
                  {

                     print_hip_main_cols (&anentry, decode);
                  }
                  else
                     print_hip_main (&anentry, decode);
                  if (read_hip_main(dataFile,&anentry)!=0)
                    break;
               }
```

```
      }

         else
            printf("%d not found\n",rec.value);
      }
    }

} /* End Main for idx_hip_main_test */
```

Each of the search programs is described briefly here.

2.2 shipmain.c

The shipmain program expects integer identifiers, i.e. HIP identifiers. After this it works in the normal way calling the search and print routines.

This program responds to the +n option by looping with the read function for the number of times specified. If EOF is reached it terminates.

2.3 shipi.c

As Section 2.2.

This record type has a main record and several transit (sub-records) so by default the transit records will be printed in tabular mode while the main record is printed verbose. Specifying `-t` will cause the main record to be printed in tabular mode but the transits will no longer be printed. Specify `-ts` to see the transits.

2.4 shipj.c

As Section 2.3.

2.5 shipep.c

As Section 2.3.

2.6 shipepe.c

As Section 2.3.

2.7 shipva.c

This works rather like `shipdm` in that it uses `find_idx_hip_va` directly, rather than a search routine. The returned index entry is then examined to see which data access routines must be used. If the `ANNEX` attribute of the index entry equals '1' then access and print routines for `hip_va_1` are called. If the `ANNEX` attribute is '2' then access and print routines for `hip_va_2` are called.

The +n option has no effect on this program.

2.8 shipdm.c

The shipdm program uses the compound hip_dm index to retrieve a requested record from one of the multiple hip_dm_? (?=C G O V or X) files. This does not call the search routine like the other programs but rather calls the `find_idx_dm` (see Section 4.2.3.12) routine directly for a given identifier. This returns an index entry. Next the routine checks the value of `hdm_idx2` of the index entry. This indicates which data access routine is used to load the record (basically a series of `if` statements are used for this).

The +n option has no effect on this program.

The -t option does not print a header in this program since a multiple retrieval may relate to different parts of the Double and Multiple Star Annex. A header is printed on `hip_dm_c` records to differentiate the two different record types (`hipdmcor` and `hipdmcom`).

2.9 shipdmc.c

The shipddmc program uses the hip_dm_c index. This is an index on CCDM identifier. The function `read_ccdm` (see Section 3.2.2.3) is used to get a CCDM identifier of the form nnnnn[+-]nnnn from the command line. A check is made that a valid identifier has been given (there may be no blanks in it). After this the search and print routines are called as usual.

Because `hip_dm_c` is stored internally as a structure containing an array of hipdmcom and an array of hipdmcor it means it has no data at the top level (it has only sub-records). In the main loop if `-t` was specified, and no other option was specified, then `decode` is set to `PRINTSUBRECS` so the data will be seen. If however the user specifies some other option this will not be done and so nothing may be seen. Hence if any other option is specified for this program then -s must be added.

The +n option has no effect on this program.

2.10 stycmain.c

The program `stycmain` checks for Tycho Catalogue identifiers in one of two formats (tyc1-tyc2-tyc3 or tyc1 tyc2 tyc3). The function `getIfTycId` returns `TRUE` if the given string yields a Tycho Catalogue identifier in the form number-number-number, it puts the identifier in the `tyc_id` structure `key`. This is called in:

```
if ( !getIfTycId(argv[i],&key) )
```

If `argv[i]` was not a Tycho Catalogue identifier then it should have been a number and there should be two more numbers following it. This can be checked using the `argc` variable, as in the line:

```
if ( argc-3 < i)
```

If there are the correct number of command line arguments then these are put into the key variable using the calls:

```
strAsINT(argv[i], &key.tyc1) ;
strAsINT(argv[i+1], &key.tyc2) ;
strAsINT(argv[i+2], &key.tyc3) ;
```

Furthermore the program will return all records for a given block if tyc2 is zero and or tyc3 is zero.

If tyc2 is not zero but tyc3 is zero then the program searches for the identifier tyc1 tyc2 1 which must exist.

If however tyc2 is zero then `find_idx_tyc_main` is used to find the first occurrence of tyc1. The `jump_tyc_main` routine is used to jump to that location and a record is read.

A loop is then entered to retrieve records until tyc1 or tyc 2 change (as appropriate).

If a partial identifier is specified then the +n option has no effect on this program. However +n is invoked if full identifiers are specified.

2.11 stycep.c

The `stycep` program works as normal except it allows for Tycho Catalogue identifiers to be specified as tyc1-tyc2-tyc3 or tyc1 tyc2 tyc3. It does not provide partial key retrieval as in `stycmain`.

A `tyc_ep` record consists of a header and all its transits. If -t is specified only the header record will be printed. The -s option must additionally be specified to print the transits.

2.12 ssolar.c

The ssolar program uses the compound index solar.idx to retrieve requested records from the files solar_ha, solar_hp and solar_t. It does not call a search routine but rather calls `find_idx_solar` (see Section 4.2.3.11) directly for a given identifier. This returns a pointer to an `idx_solar` entry. If any of the attributes `HP`,`HA` or `T` of the index entry have a value greater then zero than there is data in the corresponding data file. Three `if` statements are used to check these values and call the appropriate read and print routines. There may be multiple entries in any or all of the files for any given identifier hence the array functions are used.

Specifying `-t` will print the data in tables with the heading on top. There may be up to three tables of data for a given identifier.

2.13 shipdmg.c

This functions as normal calling `search_hip_dm_g` with keys from the command line.

This program responds to the +n option by looping with the read function for the number of times specified. If EOF is reached it terminates.

2.14 shipdmo.c

This functions as normal calling `search_hip_dm_o` with keys from the command line.

This program responds to the +n option by looping with the read function for the number of times specified. If EOF is reached it terminates.

2.15 shipdmv.c

This functions as normal calling `search_hip_dm_v` with keys from the command line.

This program responds to the +n option by looping with the read function for the number of times specified. If EOF is reached it terminates.

2.16 shipdmx.c

This functions as normal calling `search_hip_dm_x` with keys from the command line.

This program responds to the +n option by looping with the read function for the number of times specified. If EOF is reached it terminates.

2.17 shiprgc.c

All reference great circle data are loaded in memory. No search routine is provided. Rather, the routine `find_hip_rgc` is used to get a pointer to a given entry. This is printed in the normal manner.

The +n option has no effect on this program.

2.18 shipva1.c

As Section 2.2.

This retrieves records only if they appear in the file hip_va_1.dat.

2.19 shipva2.c

As Section 2.2.

This retrieves records only if they appear in the file hip_va_2.dat.

2.20 sdmsao.c

The dmsa_o.doc file may contain multiple entries for a given reference number. The search function `search_dmsa_o` (see Section 3.2.5.1) expects an array argument to deal with this. The entire file is loaded in memory so rather than returning a file pointer the search routine for `dmsa_o` returns a pointer to the first element found for the requested identifier. Since an array is returned from the search the `print_array` functions are called

instead of the normal record print functions. Identifiers are integer numbers for `dmsa_o` which are unique reference numbers.

The +n option has no effect on this program.

The options -cbs have no effect on this program.

2.21 shpauth.c

The identifier for notes (other than `dmsa_o`) is an integer, i.e. a HIP identifier.

Again there may be multiple notes for a given identifier so the `search_` function requires an array parameter. Likewise `print_array` functions are called for the result.

The +n option has no effect on this program.

The options -cbs have no effect on this program.

2.22 shdnotes.c

As in Section 2.21

2.23 shgnotes.c

As in Section 2.21

2.24 shpnotes.c

As in Section 2.21

3. Datafile data structures

The intention of this code is to provide a complete structure representing each data item as described in Volume 1. The field names in the data structures correspond to the names in Volume 1. Additionally, related information is also loaded in the structure (e.g. the notes).

For each data file (.dat file) there is C code to load and print a record of that type as well as search for a record given an appropriate identifier. The code file has the same name as the data file to make it easy to find. There are both '.c' and '.h', files where the '.h' file contains the structure and all prototypes for the file, and the '.c' file implements the functions. Hence the following files may be found (with extensions '.c' and '.h'):

```
hip_dm_c
hip_dm_g
hip_dm_o
hip_dm_v
hip_dm_x
hip_ep
hip_ep_c
hip_ep_e
hip_i
hip_j
hip_main
hip_rgc
hip_va_1
hip_va_2
tyc_main
tyc_ep
solar_ha
solar_hp
solar_t
```

Routines are also provided for the notes files. These are:

```
dmsa_o
hg_notes
hd_notes
hp_notes
hp_auth
```

Additionally there are auxiliary data structures which are used to hold sub-records contained within the data file these are:

`ccdm`	stores CCDM identifier as separate number
`hepetran`	hip_ep_e transit record
`hipdmcom`	hip_dm_c component record
`hipdmcor`	hip_dm_c correlation record
`hipjtran`	hip_j transit record
`hipoint`	hip_j pointing record
`hiptrans`	hip_ep transit record
`tyc_id`	stores tyc_id as three integers
`tyctrans`	tyc_ep transit record
`hipabsc`	hip_i abscissa record

3.1 Header file information (data files)

In any given header file there are the following:

- a constant defining the length of one record of this type in the text file. This has '_REC_LEN' appended to the name of the file in lower case. Hence for hip_main there is hip_main_REC_LEN. Note that the record length here is taken to include the cr and lf at the end of the record.
- any required include files. For example, the index used for this data file will be included as well as any other structures used, such as transit structures.
- a structure with the same name as the data file which contains one attribute for each field in the file. The attributes are named as outlined in Volume 1. Hence for hip_main a 'struct hip_main' definition is provided with attributes H0 to H77.
  Attribute types can be CHAR, INT, FLOAT, BITS, NONLINEAR (defined in utils.h), ccdm or tyc_id (defined in ccdm.h and tyc_id.h).
  Additionally, if there is some related information such as a set of transits which go with this record, there will be an array defined to hold this information. The array definition will be found in the header file of the same name without 'array_' in front of it. Hence the definition of array_hip_trans will be found in the file hip_trans.h. This file will have been included above.
- a typedef for the struct so the name of the struct may be used directly elsewhere in the code. Thus to define a variable of type 'struct hip_main' it is only necessary to have 'hip_main myvar;'.
- a print prototype which has 'print_' prepended to the name, e.g. for hip_main there is print_hip_main. It prints the attribute name and value of each attribute of the structure pointed to by entry one per line to standard output.
- a print prototype which has 'print_' prepended and '_cols' appended to the name, e.g. for hip_main there is print_hip_main_cols. This prints the data across the screen in one row, as it would have appeared in the original data file.
- a read prototype which has 'read_' prepended to the name, e.g. for hip_main there is read_hip_main. The read function reads a record from the given file pointer (fp) and places the data in the struct pointed to by entry.
- a search prototype which has 'search_' prepended to the name, e.g. for hip_main there is search_hip_main. This uses the index to find a record given a key value.
- a jump prototype which has 'jump_' prepended to the name. This takes a record number and returns a file pointer to the location in the file where that record may be found.

Here is an example of a header file (tyc_ep.h):

```
/*  Hipparcos ASCII CD-ROM load and search routines Release 1.1 June 1997
    William O'Mullane
    Astrophysics Division, ESTEC, Noordwijk, The Netherlands.
    See the readme.pdf file for more information */
#ifndef _tyc_ep_H_
#define _tyc_ep_H_
#define tyc_ep_REC_LEN 86

#include "itycep.h"
#include "tyctrans.h"

struct tyc_ep
{
     INT       TH1;
     INT       TH2;
     INT       TH3;
     INT       TH4;
     INT       TH5;
     FLOAT     TH6;
     FLOAT     TH7;
     FLOAT     TH8;
     FLOAT     TH9;
     INT       TH10;
     FLOAT     TH11;
     FLOAT     TH12;
     FLOAT     TH13;
     FLOAT     TH14;
     BITS      TH15;
     BITS      TH16;
     array_tyctransTRANSITS;
}; /* End Struct */

typedef struct tyc_ep tyc_ep ;


int print_tyc_ep (tyc_ep* entry, int decode) ;
int print_tyc_ep_cols (tyc_ep* entry, int decode) ;
int print_tyc_ep_header () ;
FILE* jump_tyc_ep (long recNum) ;

int read_tyc_ep (FILE* fp,tyc_ep* entry) ;
FILE* search_tyc_ep (tyc_id* key, tyc_ep* arecord) ;
#endif /* _tyc_ep_H_ */
```

3.1.1 Arrays

In addition to the above, if this structure is used in an array, there will be:

- a constant which is the name with 'array_' prepended and '_size' appended to it, e.g. for hip_trans there will be array_hip_trans_size. This defines the maximum number of elements allowed in the array.

- a structure which has 'array_' prepended to the name, e.g. for hip_trans there is array_hip_trans. This structure contains two members:
  - an integer no_entries which contains the number of entries in the array.
  - an array data which is of the type of the name (in this case hip_trans) and has the size defined by the constant array_'name'_size.
- a typedef for the struct.
- a print prototype which has 'print_array_' prepended to the name. This is to print the elements of the array in verbose mode.
- a print prototype which has 'print_array_' prepended and '_cols' appended to the name. This is to print the elements of the array in tabular mode.
- a read prototype which has 'read_array_' prepended to the name.

3.1.2 tyc_id.h

This is very similar to the other header files, but does not relate directly to a data file. Some files contain Tycho Catalogue identifiers as 3 digits in a string. When such fields are loaded they are split into 3 numbers and put in a tyc_id. Thereafter tyc1 tyc2 tyc3 may be accessed for this field. Field T1 of TYC_MAIN is an example, where T1.tyc1 etc. may be used to access its components.

Hence tyc_id.h may be included as necessary in other files where tyc_id is used.

Print and read prototypes are provided for tyc_id. However the read prototype expects to be passed a string (field) which contains the Tycho Catalogue identifier and a pointer to a tyc_id structure (entry) into which it puts the retrieved numbers.

A special function getIfTycId is also provided which parses a string to determine whether it is a Tycho Catalogue identifier expressed as tyc1-tyc2-tyc3. This is used in the search program to allow identifiers to be expressed as tyc1-tyc2-tyc3 or tyc1 tyc2 tyc3, see Section 2.10.

3.1.3 ccdm.h

This is precisely the same as tyc_id but is used to store CCDM identifiers as two numbers (left and right), and a sign (see also Section 4.2.3.10). This is used for example in hipdmcom for field DC1.

3.2 C file information

The C files contain the code for the read, print, search and jump functions which are described here. Additionally it may contain a lookup table for decoding a bit field.

3.2.1 Bit field lookup tables

Some fields in the catalogue are encoded bit fields. A function is provided in utils called 'showFlags' (see Section 5.3.16) which accesses a bit field and prints the strings from the given lookup table if the equivalent bits are set.

The array of strings has meaning for only one field, and is defined in the appropriate code files, thus hiptranc.c has HT4_codes, tyc_ep.c has TH15_codes and TH16_codes, and tyctrans.c has TT13_codes.

3.2.2 Read functions

There are several different types of read function depending on the type of data being dealt with.

3.2.2.1 Read function

The read function expects two arguments:

- `FILE* fp` - an open file pointer to the data file containing the record to be read.
- `'data_type'* entry` - where data_type is the `struct` of the correct type for the record to be read. The argument `entry` must point to an allocated structure which will be filled with the data from the read record.

The function assumes the file pointer passed is already positioned at a correct record boundary. This is done for example by calling `jump_'data_type'`.

Next the individual fields are extracted. The standard C function `strtok` is used for this. The `strtok` function splits a string into individual tokens based on some field delimiter, in this case '|'. Note '\r' is also a field delimiter because it is the end of record marker. The function remembers the string it is working on and each successive call to it with `NULL` returns the next token from the string. Each token is retrieved and the correct conversion is applied to it so it may be stored in the correct attribute of the structure pointed to by `entry`.

If any attributes are arrays then they are also loaded using the appropriate `read_array` function.

Some records e.g. `hip_ep`, are made up of multiple lines in the file. The read function also takes care of this by reading the next line as necessary.

3.2.2.2 Read function for tyc_id

The function `read_tyc_id` is a special case as it is read from a compound field of three numbers. It expects two arguments:

- `char* field` - a string containing the three number Tycho Catalogue identifier.
- `tyc_id* entry` - the `tyc_id` to put the number into.

This simply uses `sscanf` to read the three numbers from the string. It also sets them all as valid.

3.2.2.3 Read function for ccdm

As for `tyc_id` this is a special case as it is read from a compound field of two numbers separated by a + or -. The function `read_ccdm` expects two arguments:

- `char* field` - a string containing the CCDM identifier.
- `ccdm* entry` - the `ccdm` structure to put the number into.

This simply uses `sscanf` to read the numbers and sign from the string. It also sets `left` and `right` as valid.

3.2.2.4 Read array function

For those datatypes which have arrays the read array function will fill a given array from the given file. Arrays are used to hold the transits for an entry. This function expects three arguments:

- `FILE* fp` - an open file pointer to the data file containing the record to be read.
- `int no_entries` - the number of records to be retrieved from the file.
- `array_'data_type'* array` - where `data_type` is the structure of the correct type for the record to be read; this is a pointer to an array to hold the records read from the file.

This loops for `no_entries` times and calls the `read_function` passing it `fp` and a pointer to a different array entry each time.

3.2.3 Jump function

This has the name '`jump_`' prepended to the datatype name. It makes use of the record length of the data being dealt with to jump to a certain location in the file given a record number. It uses `fseek` to achieve this. It returns the file pointer so a read may then be performed using this pointer. It expects one argument:

- `long recNum`: number of record to jump to.

This function opens the data file the first time it is called and then stores the file pointer in a static variable for future use.

3.2.4 Print functions

3.2.4.1 Verbose Print function

This function has '`print_`' prepended to the name, e.g. `print_hip_main`. The print function expects two arguments:

- `'data_type'* entry` - a pointer to the structure to be printed.
- `int decode` - which is used to decide if bit fields are decoded and if sub-records are printed. The values for `decode` are defined in utils.h.

This function has a `printf` statement for each attribute of the structure which prints the attribute name and value on a line. It also prints a description of the attribute on the same line, as specified in Volume 1 of the printed catalogue. This is referred to in the text as verbose printing of a record and is the default used in the search programs.

Arrays are printed in tabular mode unless `ARRAYVERBOSE` is contained in `decode` in which case arrays are printed verbose. The value of `decode` is passed to the `print_array_` function.

The `showFlags` function is called for any fields of type `BITS`.

The function `print_NONLINEAR_cols` is called for any fields of type `NONLINEAR`.

3.2.4.2 Print as columns function

This function has 'print_' prepended and '_cols' appended to the name of the structure, e.g. print_hip_main_cols. It prints the data as it would have appeared in the data file originally. This also takes two arguments:

- 'data_type'* entry - a pointer to the structure to be printed.
- int decode - which is used to decide if bit fields are decoded and if sub-records are printed. The values for decode are defined in utils.h.

This function calls the conversion to string (NAMEasStr) routine for each attribute and prints the resulting string with '|' appended. For the last attribute '/r/n' is appended in place of '|'.

For arrays it also prints the contents of the arrays if decode equals DOALL or if PRINTSUBRECS is contained in the decode value.

For bit fields if decode is set to DOALL or if DECODEBITS is contained in the decode value, the showFlags function is called to print flag settings for the bit field.

For nonlinear encoded fields (correlation coefficients) if decode is set to DOALL or if EXTRACTCOEFF is contained in decode the print_NONLINEAR_cols is called for that field instead of the normal NONLINEARasStr function. This means that instead of printing all numbers concatenated they are printed with a space between each number.

3.2.4.3 Print header function

This function which has 'print_' prepended and '_header' appended to the name of the structure, e.g. print_hip_main_header may be used to print a banner above the output from one of the print_'NAME'_cols function. It takes no arguments.

3.2.5 Search functions

The search function associates a particular index find routine with the dataset and uses this index to look up records in the file. The reason it is placed in this file and not in the index file is that the same index may be used for different data files.

The search function expects two arguments:

- idx_'data_type'* key - a pointer to the key value to be searched for.
- 'data_type'* arecord - a pointer to the structure to store the retrieved record in.

The find routine is used to search the index for the key required and return a record number.

The jump routine is used to jump to the correct location in the data file for the given record number.

The returned file pointer is passed to the read function to retrieve the correct record. The file pointer is the returned and may be used to read the next record as is done in several of the test routines provided. This varies slightly in some routines depending on the return type of the index. Each routine is briefly described here.

3.2.5.1 dmsa_o search_dmsa_o (INT* key, array_dmsa_o* array);

Since dmsa_o.doc is small it is loaded in its entirety into memory as an array. For any given HIP identifier there will be multiple consecutive dmsa_o entries. The array_dmsa_o structure contains a pointer to a dmsa_o entry and a no_entries attribute and may be used as any other array.

This function uses the find_ dmsa_o routine to get a pointer to the first dmsa_o entry for this HIP identifier. The field DM02 contains the number of entries for this identifier so this is assigned to no_entries of the array.

Since this does not deal with a file, no file pointer is returned. Instead a pointer to the first entry in the array is returned.

3.2.5.2 FILE* search_hd_notes (INT* key, array_hd_notes* array);

Again there are multiple entries per identifier for hd_notes so it also returns an array. It uses find_ to find the record number of the first entry in the file, read_array is used to read all entries for this identifier into the array provided.

It uses jump_ to jump forward to the position after these entries and returns the file pointer as in the other search functions.

3.2.5.3 FILE* search_hp_notes (INT* key, array_hp_notes* array);

As Section 3.2.5.2

3.2.5.4 FILE* search_hg_notes (INT* key, array_hg_notes* array);

As Section 3.2.5.2

3.2.5.5 FILE* search_hip_dm_c (ccdm* key, array_hip_dm_c* array);

As Section 3.2.5.2

Note that hip_dm_c has its own index on CCDM identifier. The other search_hip_dm_? (? = G O V or X) routines all use a common index (hip_dm.idx).

3.2.5.6 FILE* search_hip_dm_g (INT* key, hip_dm_g* arecord);

This file uses the hip_dm compound index. It first calls find_idx_hip_dm with the key. This returns a pointer to an idx_hip_dm index entry or NULL if it is not found.

Next the routine checks that, if an index entry was found, that hdm_idx2 (column 2) is 'G'. If this is the case a hip_dm_g entry exists. The jump_ routine is used to jump the data file to the correct position, then the read_ routine is called to read the actual record. If it is successful the file pointer is returned.

3.2.5.7 FILE* search_hip_dm_o (INT* key, hip_dm_o* arecord);

This works precisely as Section 3.2.5.6 except that the value of hdm_idx2 should be 'O' here.

3.2.5.8 FILE* search_hip_dm_v (INT* key, hip_dm_v* arecord);

This works precisely as Section 3.2.5.6 except that the value of `hdm_idx2` should be 'V' here.

3.2.5.9 FILE* search_hip_dm_x (INT* key, hip_dm_x* arecord);

This works precisely as Section 3.2.5.6 except that the value of `hdm_idx2` should be 'X' here.

3.2.5.10 FILE* search_hip_va_1 (INT* key, hip_va_1* arecord);

This routine uses the shared hip_va index (common to hip_va_1 and hip_va_2) hence it uses `find_idx_hip_va` (see Section 4.2.3.13) which returns a pointer to an `idx_hip_va` entry.

If the entry exists and if the `ANNEX` attribute is '1' then it uses `jump_hip_va_1` to jump to the record number indicated in the index entry (`RECNO.value`). It next reads the record at that location. If it is successful it returns the file pointer.

3.2.5.11 FILE* search_hip_va_2 (INT* key, hip_va_2* arecord);

As Section 3.2.5.10 except `ANNEX` is checked for value '2'.

3.2.5.12 FILE* search_hp_auth (INT* key, array_hp_auth* array);

The hp_auth index is different to the other indices, being more of a relational table (see Section 4.2.3.14). The call to `find_idx_hp_auth` returns a pointer to the first entry for the given HIP identifier (`key`).

This routine then builds an array of `hp_auth` records by examining each index entry for this HIP identifier. For the index entry it extracts the `hp_auth` record number (`RECNUM` attribute of the index entry) which it then uses in conjunction with `jump_hp_auth` to position the file pointer correctly before reading that entry. It knows it has finished when the `HIPID` (attribute of the index entry) changes or when there is a read error in the file.

3.2.5.13 FILE* search_tyc_main (tyc_id* key, tyc_main* arecord);

The tyc_main index is based on the record location of the first entry for each tyc1 in the data file. Hence looking at the index just gives a place to start looking in the file. The next location in the index gives a bound not to go beyond when searching. In between these boundaries a binary search may be performed.

Unlike other search routines `search_tyc_main` loads the index locally and stores it in a static array. It does not use the `find_idx_tyc_main` function.

The given `key` is checked so that tyc1 is not greater than `idx_tyc_main_no_entries` and not less than 1, tyc2 is checked so that it is greater than 1, finally tyc3 is checked to be between 1 and 8 (inclusive).

Next the left and right boundaries are set up using the index. The number:

```
index[key->tyc1.value-1].value
```

(value in the array at tyc1 -1) will give the address of the first entry in tyc_main.dat for this tyc1. Similarly:

```
index[key->tyc1.value].value
```

(value in the array at tyc1) will give the address of the first entry of the next tyc1 or the right bound of the search.

Next a binary search loop is entered where arecord is read from tyc_main until a match is found for tyc2.

If a match is found for tyc2 then one last jump is required to find the match for tyc3. This is straightforward as tyc3 increases sequentialy without gaps for a given tyc1 and tyc2. Hence a record can be found relative to the current record using its tyc3 and the requested tyc3 as follows:

```
mid - arecord->T1.tyc3.value + key->tyc3.value;
```

where mid is the record number of the current record.

Finally the above will always yield a record but it may not be the one requested so this is checked before returning a positive result.

3.2.5.14 FILE* search_hip_ep (idx_hip_ep* key, hip_ep* arecord);

This is standard as outlined in Section 3.2.5.

3.2.5.15 FILE* search_hip_ep_e (idx_hip_ep* key, hip_ep_e* arecord);

This is standard as outlined in Section 3.2.5.

3.2.5.16 FILE* search_hip_i (idx_hip_i* key, hip_i* arecord);

This is standard as outlined in Section 3.2.5.

3.2.5.17 FILE* search_hip_j (idx_hip_j* key, hip_j* arecord);

This is standard as outlined in Section 3.2.5.

3.2.5.18 FILE* search_hip_main (idx_hip_main* key, hip_main* arecord);

This is standard as outlined in Section 3.2.5.

3.2.5.19 FILE* search_tyc_ep (tyc_id* key, tyc_ep* arecord);

This is standard as outlined in Section 3.2.5.

4. Index files

For each index file (.idx file) there is C code to load the index and find an index entry given an appropriate identifier. The code file has the same name as the index file but with the letter 'i' prepended and all underscores removed. Again there are both '.c' and '.h' files, where the '.h' file contains the structure and all prototypes for the file, and the '.c' file implements the functions. Hence the following files may be found (with extensions '.c' and '.h')

```
ihipmain
itycmain
ihipdm
ihipdmc
ihipep
ihipi
ihipj
ihipva
isolar
itycep
```

Index code exists for the following '.idx' files in the notes directory:

```
ihpauth
ihdnotes
ihgnotes
ihpnotes
```

4.1 Header files information (index files)

In any given header file the following are given:

- a constant defining the length of one record (index entry) of this type in the text file. This has '`_REC_LEN`' appended to the name of the file in lower case. Hence for `idx_hip_main` it will be `idx_hip_main_REC_LEN`.
- a constant defining the number of entries expected in the index. This is used as a check when loading the index as well as for defining the array to hold the index. This has '`_entries`' appended to the name of the file in lower case. Hence for `idx_hip_main` there is `idx_hip_main_entries`.
- possibly a structure with the same name as the data file which contains one attribute for each field in the file. This only applies if the index has multiple fields.
- a `typedef` for the struct so the name of the struct may be used directly elsewhere in the code.
- a 'find' prototype which has '`find_`' prepended to the name of the data file, e.g. for `idx_hip_main` there is `find_idx_hip_main`. The find function takes a key and an index and looks up the key in the index and returns a record number.
- a 'load' prototype which has '`load_`' prepended to the name which takes a file name (for the index) and a pointer to a structure to hold the index. It then loads the index from the file into the memory structure.
- a 'getnext' prototype which has '`getnext_`' prepended to the name which takes a pointer to an opened file and a pointer to an element to hold one index entry. It then loads the next index entry from the file into the memory structure.

Again here is an example file (itycep.h):

```
/*  Hipparcos ASCII CD-ROM load and search routines Release 1.1 June 1997
    William O'Mullane
    Astrophysics Division, ESTEC, Noordwijk, The Netherlands.
    See the readme.pdf file for more information */
#ifndef _idx_tyc_ep_H_
#define _idx_tyc_ep_H_
#define idx_tyc_ep_REC_LEN 23

#include "tyc_id.h"

struct idx_tyc_ep
{
     tyc_id    tycep_idx1;
     INT       tycep_idx2;
}; /* End Struct */

typedef struct idx_tyc_ep idx_tyc_ep ;


#define idx_tyc_ep_entries  34446

long find_idx_tyc_ep (tyc_id* key);
int load_idx_tyc_ep (char* ifile, idx_tyc_ep* index) ;
int getnext_idx_tyc_ep (FILE* fp, idx_tyc_ep* entry);
#endif /* _idx_tyc_ep_H_ */
```

4.2 Functions

The C files contain the code for the 'getnext', 'load', and 'find' functions which are described here. The prototypes are contained in the header file.

4.2.1 load function

The load function expects two arguments:

- `char* ifile` - a string containing the name of the index file.
- `idx_'data_type'* index` - a pointer to the array which will contain the index.

This function attempts to open the index file.

Upon successful opening of the file the function loops for the number of entries that should be in the index file and calls the `getnext` function passing the file pointer and a pointer to the appropriate array element for that index entry.

It reports an error if the file does not contain enough entries.

4.2.2 getnext function

The load function expects two arguments:

- `FILE* fp` - a file pointer to the open index file.

- `idx_'data_type'* idx_entry` - a pointer to a data structure of the correct type to hold one index entry.

One record from the current file position is read into a string buffer. This is then converted to the appropriate data values for storage in the data structure pointed to by `idx_entry`. The function `strtok` may be used for this (see also Section 3.2.2).

4.2.3 find function

The find function expects one argument:

- `'identifier_type'* key` - a pointer to the key value to be searched for.

There are two static variables defined in this function:

- `int init` - used to check if the function has already been called once.
- `idx_'data_type' index[idx_'datat_type'_no_entries]` - an array which contains the index.

The find function looks up the index in the correct way to return a record number for the given key value. This is normally some variation of a binary search. The first time it is called it loads the index using the appropriate load routine.

In the case of `ihipdm`, `ihipva`, `ihpauth`, `isolar` a pointer to an actual index entry is returned. This is because the index is more complex. Each find function is briefly described here.

4.2.3.1 long find_idx_hip_main (idx_hip_main* key);

Because there is almost a full set of HIP identifiers there is entry in the index for every HIP identifier. If no record exists for a given HIP identifier the value '-1' is stored in the index file. This file is loaded in the `index` using the `load_idx_` routine. Arrays in C start at zero whereas HIP identifiers start at one, so the record number for a particular HIP identifier (say n) is found in the array at position n-1.

This returns the value in the array at the position specified by (`key->value -1`).

4.2.3.2 long find_idx_hip_ep (idx_hip_ep* key);

As for Section 4.2.3.1

4.2.3.3 long find_idx_hip_i (idx_hip_i* key);

As for Section 4.2.3.1

4.2.3.4 long find_idx_hip_j (idx_hip_j* key);

As for Section 4.2.3.1

4.2.3.5 long find_idx_tyc_main (idx_tyc_main* key);

The tyc_main index works on tyc1 only. Hence for a given tyc1 this indicates the record number in the file for the first entry for that tyc1 value. Again an entry exists in `index` for every value of tyc1 so the record number for a given value of tyc1 (say n) if found in the

array at location n-1. After finding this a binary search must be performed as explained in Section 3.2.5.13.

4.2.3.6 long find_idx_hd_notes (INT* key);

The file hd_notes.idx contains two columns: one for the HIP identifier and one for the record number of the first note for this identifier. An index entry then has the following structure:

```
struct idx_hd_notes
{
     INT       HIPID;
     INT       RECNO;
}; /* End Struct */
```

A simple binary search is therefore done on `HIPID` in `index`. Not all HIP identifiers appear in the index so the right side of the search may be bounded to be the given HIP identifier (`key->value`) i.e. for HIP identifier n it must appear before location n in the array.

If the identifier is found `RECNO.value` is returned for that index element.

4.2.3.7 long find_idx_hg_notes (INT* key);

As Section 4.2.3.6

4.2.3.8 long find_idx_hp_notes (INT* key);

As Section 4.2.3.6

4.2.3.9 long find_idx_tyc_ep (tyc_id* key);

The tyc_ep.idx has a tyc identifier and record number of the entry for that identifier in tyc_ep.dat. Internally tyc identifiers are stored in the tyc_id struct:

```
struct tyc_id
{
     INT       tyc1;
     INT       tyc2;
     INT       tyc3;
}; /* End Struct */
```

The index entries are loaded into the following structure:

```
struct idx_tyc_ep
{
     tyc_id    tycep_idx1;
     INT       tycep_idx2;
}; /* End Struct */
```

This means a binary search may be performed to find a match on tyc1 against `tycep_idx1`. It is now uncertain if the current entry is in the middle of that block or not presenting two possibilities:

Go Back: search back in the array looking for a match for tyc2 if the requested tyc2 is less than the one being interrogated. If a match on tyc2 is found search further back to find a match on tyc3.

Go Forward: the requested tyc2 is greater than the one being interrogated so search forward in the array to find a match. If it is found search forward again for a match for tyc3.

The value of `tycep_idx2` is returned from the function if the requested identifier is matched.

4.2.3.10 long find_idx_hip_dm_c (ccdm* key);

The hip_dm_c.idx file has two columns, the first contains a CCDM identifier and the second contains the record number of the first record for this solution.

The CCDM identifier is stored as a 3 part structure:

```
struct ccdm
{
     INT       left;
     CHAR      sign[2];
     INT       right;
}; /* End Struct */
```

The index entry has the following structure:

```
struct idx_hip_dm_c
{
     ccdm      dmc_idx1;
     INT       dmc_idx2;
}; /* End Struct */
```

A binary search is performed on the `dmc_idx1.left` for `key->left` in the `index`. If it is found the routine backs up the array to the first entry for `key->left`. It then searches forward to find a match for the `key->right` and `key->sign`. It returns `dmc_idx2.value` if a match is found.

4.2.3.11 idx_solar* find_idx_solar (INT* key);

The solar.idx file serves as compound index for the three files solar_ha, solar_hp and solar_t. It has eight columns which are loaded into the idx_solar struct:

```
struct idx_solar
{
     INT       NUM;
     INT       SEQ_NUM;
     INT       HA;
     INT       HA_NUM;
     INT       HP;
     INT       HP_NUM;
     INT       T;
     INT       T_NUM;
}; /* End Struct */
```

`NUM` is the solar system object identifier. `SEQ_NUM` is the sequence number (1-55). `HA` is the offset in solar_ha.dat of the first entry for this identifier (or -1 if none) and `HA_NUM` is the number of entries for this identifier in the file. Likewise `HP, HP_NUM, T` and `T_NUM` contain this information for the files solar_hp and solar_t.

A binary search is carried out on NUM to find key. If a match is found a pointer is returned to the index entry so that the calling function may make use of the complex index in whatever way it wishes.

For an example of the usage of this index see ssolar.c.

4.2.3.12 idx_hip_dm* find_idx_hip_dm (INT* key);

The hip_dm.idx is a compound index for the various hip_dm_? (? = C G O V or X) files. It has three columns which are loaded into the following structure:

```
struct idx_hip_dm
{
     INT       hdm_idx1;
     CHAR      hdm_idx2[2];
     INT       hdm_idx3;
}; /* End Struct */
```

hdm_idx1 contains the HIP identifier, hdm_idx2 contains a character (C G O V X) corresponding to the hip_dm_? file which contains the solution for this identifier. hdm_idx3 contains the offset at which the solution for this identifier may be found.

A binary search is performed on hdm_idx1 to find key. If a match is found then a pointer is returned to the index entry so the calling function may make use of the information in it.

For an example of how to use this information see shipdm.c.

4.2.3.13 idx_hip_va* find_idx_hip_va (INT* key);

The hip_va.idx file serves as an index for hip_va_1 and hip_va_2 data files. The index contains three columns which are loaded into a structure:

```
struct idx_hip_va
{
     INT       HIPID;
     INT       ANNEX;
     INT       RECNO;
}; /* End Struct */
```

HIPID contains the HIP identifier. ANNEX will have the value 1 or 2 indicating which data file the entry is in. RECNO is the record number of this entry in the datafile.

A binary search is done on HIPID and a pointer is returned to the index entry so it may be used in the calling function.

For an example of using this see shipva.c.

4.2.3.14 idx_hp_auth* find_idx_hp_auth (INT* key);

This index contains two columns HIPID and RECNUM. The record number is the record number in the file hp_auth.dat. A specific feature of this index is that the HIPID may repeat itself, hence this is more of a relational lookup table than an index.

This find routine then loads the index the first time it is called, then does a binary search to find the HIP identifier required. Since there may be multiple HIP identifiers this may select an entry in the middle of the list. Hence upon finding the required identifier the

function steps back up the array until it finds the entry before the first entry for this `HIPID`. It the returns a pointer to the first entry for the required identifier.

5. utils

The utils.h and utils.c files describe functions, structures and constants used by every file in the system. Capitals have been used for constants and for structures which are used as data types for the Hipparcos and Tycho data.

These structures and functions are used by all of the load routines, hence if portability problems are encountered it should only be necessary to modify utils.

5.1 General definitions

Initially in utils.h some constants and simple `typedef` definitions are made as follows:

- `TRUE` and `FALSE` are defined as `1` and `0`.
- `ASRAW ARRAYVERBOSE DECODEBITS PRINTSUBRECS EXTRACTCOEFF` and `DOALL` are defined for use in the decode parameter of the print functions.
- `EMPTYBITSTRING` and `BITSTRLEN` are defined for use when dealing with `BITS`.
- `bool` is defined as an `int` (using `typedef`)
- `CHAR` is defined as `char` (using `typedef`)

5.2 Definition of structures

All structures are describe here. Additionally for each structure there is a `typedef` which allows the name of the structure to be used in the code, e.g. '`INT H1;`' would appear in the code not '`struct INT H1;`' (the second statement is equally valid). For each structure there always exist two routines for conversion to and from a string. These always have the names `strAsNAME` (string As NAME) and `NAMEasStr` (NAME as String), e.g. for INT you will find `strAsINT` and `INTasStr`.

5.2.1 struct NONLINEAR

The `NONLINEAR` structure is used to hold parameters which are strings of integers in nonlinear encoding. The functions `strAsNONLINEAR` and `NONLINEARasStr` are used to read and print the structure. The function `crackStr` breaks a string of concatenated integers up and places them in an array. Functions `print_NONLINEAR` and `print_NONLINEAR_cols` are also provided.

The `NONLINEAR` structure has four attributes:

- `bool isValid`: this is set if the value is valid. If when reading the value the field was blank then this would be set to `FALSE`.
- `int noEntries`: the number of entries found in the coded field.
- `char field[200]`: the original input string.
- `int value[NONLINEAR_ARSIZE]`: an array of integers containing the numbers extracted from the coded string.

5.2.2 struct BITS

The BITS structure is used for fields in the data files which are binary encoded integers. The functions strAsBITS and BITSasStr are used to create this from text and to return the text version.

The BITS structure has three attributes:

- bool isValid: this is set if the value is valid. If when reading the value the field was blank then this would be set to FALSE.
- int value: the integer value as read from the data file.
- char bitString[BITSTRLEN]: this array is used to hold the bit values provided by the routine intToBits (defined in utils.h).
  Note that intToBits fills this array such that the value of the bit corresponding to 2^0 is in position zero. This makes checking if a particular bit is set very simple, e.g. to check 2^4 just reference location 4 in the array (i.e. bitString[4]). This also means that when printing this array it appears the opposite way around for a binary number (which will normally have the bit corresponding to 2^0 at the rightmost position).

5.2.3 struct INT

The INT structure is used to hold integer fields. Again there are functions to convert this to and from a string. These are strAsINT and INTasStr.

The INT struct has two attributes:

- bool isValid: this is set if the value is valid. If when reading the value the field was blank then this would be set to FALSE.
- int value: the integer value as read from the data file and converted using atoi.

5.2.4 struct FLOAT

The FLOAT structure is used to hold floating point fields. Again there are functions to convert this to and from a string. These are strAsFLOAT and FLOATasStr.

The FLOAT struct has three attributes:

- bool isValid: this is set if the value is valid. If when reading the value the field was blank then this would be set to FALSE.
- double value: the floating point value as read from the data file and converted using atof.
- int precision: the real precision of this number, i.e. the number of significant places after the decimal.

5.3 Functions

In the header file the prototypes for the following functions may be found. They are implemented in utils.c.

5.3.1 int NONLINEARasStr(char* str, char* format, NONLINEAR* nums);

Return the original encoded string in str.

5.3.2 int BITSasStr(char* str, char* format, BITS* num);

Return the original integer formatted as specified in `format` in the string `str`.

5.3.3 int INTasStr(char* str, char* format, INT* num);

Return the integer formatted as specified in `format` in the string `str`.

5.3.4 int FLOATasStr(char* str, char* format, FLOAT* num);

Return the floating point number formatted as specified in `format` in the string `str`.

This function takes the assigned precision into account and will buffer to the right with blanks rather than '0' as necessary. For example, if the given number has precision three its value is 8.930 and the requested format is %6.4f, then the string '8.930_' will be returned and not '8.9300'.

If there is insufficient space in the format for a 0 in front of a decimal number then the preceding zero is dropped. For example given -0.45 with format %4.2f the string '-.45' will be returned.

When a sign is always specified i.e. + is the first character in the format. The sign is only output if the value is non zero.

5.3.5 void strAsNONLINEAR (char* str, NONLINEAR* nums);

Given the string `str` break it up using `crackStr` into a set of integers where each three characters represent a number. The numbers are stored in the `value` attribute of `nums`. If the string is not blank then `nums->isValid` is set.

5.3.6 void strAsBITS (char* str, BITS* num);

Convert `str` to a bit string (using `atoi` and `intToBits`) and store the string and the original number in the `BITS` structure `num`. If the string is not blank then `num->isValid` is set.

5.3.7 void strAsINT (char* str, INT* num);

Convert `str` to an integer (using `atoi`) and store it in the `value` attribute of the INT struct `num`. If the string is not blank then `num->isValid` is set.

5.3.8 void strAsFLOAT (char* str, FLOAT* num);

Convert `str` to a floating point number (using `atof`) and store it in the `value` attribute of the `FLOAT` `num`. If the string is not blank then `num->isValid` is set.

This function checks to find the number of significant digits after the decimal and stores this number in the `precision` attribute of `num`.

5.3.9 void print_NONLINEAR(NONLINEAR* nums);

Prints the values contained in `nums` one per line with the array reference.

5.3.10 void print_NONLINEAR_cols(NONLINEAR* nums);

Prints the values in `nums` across the screen with a new line after every 20 numbers.

5.3.11 bool isBlank (char* str);

Checks if a string is blank and returns `TRUE` if it is.

5.3.12 int getWidth (char* format);

Looks at a format string (e.g. %11.3f) and returns the width specified in it (i.e. 11 in this case).

5.3.13 int getPrecision (char* format);

Looks at a format string (e.g. %11.3f) and returns the precision specified in it (i.e. 3 in this case).

5.3.14 bool blankOut (char *str, int width);

Fills a string with `width` blank spaces. This is necessary for printing empty fields.

5.3.15 bool intToBits(char* bits, int num);

Converts the integer `num` into a bit string `bits`.

5.3.16 void showFlags(char* flags[], BITS* bits);

The `flags` argument here must be an array of strings which correspond to the meaning of a particular bit being set in the bitstring. So the first string should correspond to the meaning of bit 0 being set. This routine simply loops through the strings and prints the string if the equivalent bit in `bits` is 1 (see also Section 3.2.1).

5.3.17 int crackStr(int* value, char* str, int width);

Takes the string `str` and breaks it into integer numbers each of exactly `width` size. It stores these in the array `value`. This function returns the number of integers retrieved from the string.

5.3.18 void checkIfCommandLineFlag (char **argv,int argc,int* i, int *cols,int *decode, int *quite, long* multiRecs);

This function is called by all applications to check for flags on the command line and print the help screen. It expects a pointer to `argv` and the value of `argc`; `i` is a pointer to an integer which is the current position on the command line; `cols`, `decode`, `quite` and `multiRecs` are all pointers to integers which may be modified by this routine. The variable `i` is also modified by this routine such that `i` will be the position of the first argument after the block of options when the function returns. The values put in the variables must be interpreted in the main loop afterwards.

Basically this function looks for '+' or '-' in the current argument and checks to see which options are specified. It loops through the `argv` doing this until it finds an argument which does not start with'+' or '-'.

If a '-' is found then the function checks for options b a s c and t. These may appear together after the '-'.

The option 'a' is intended to allow specification of b c s. It sets decode to `DECODEBITS + PRINTSUBRECS + EXTRACTCOEFF`.

The option 'b' is intended to turn on bit decoding. The bit string is actually decoded when the number is loaded; so this is changing the way it is printed out to show the decoded strings for each bit set. It adds `DECODEBITS` to the variable decode.

The option 'c' is intended to turn on extraction of correlation coefficients. This adds `EXTRACTCOEFF` to decode. Again the coefficients are actually extracted when the record is loaded, and this only affects the way they are printed out.

The option 's' ensures printing of sub-records. This adds `PRINTSUBRECS` to decode.

The option 't' prints the main record in tabular instead of verbose mode. This sets `cols` to the value `1`.

The option 'q' is intended to stop the printing of the header when -t is specified. For `shipmain`, `stycmain`, specifying `-tq` will give output identical to the original datafile.

A default option (any other letter) is defined which prints the help message (if h was not specified it also prints an invalid option warning). If the default is invoked the routine exits the program.

Because some options simply cause values to be added to decode specifying an option twice will cause decode to assume a value which the print functions will no longer be able to interpret.

6. Compilation

The src directory must first be copied to the local disk. A Makefile is provided for all code in src. Hence one need only type:

```
make
```

in the src directory to compile all of the provided files. Ensure the environment variable CC points to the preferred compiler (if it is not set it will run cc by default). If a sequence of errors are given the compiler is probably not ANSI compliant and it may be necessary to use another compiler or pass some extra flags to the existing one. The gnu compiler gcc is an ANSI compliant compiler which is free and available on most platforms (including DOS).

6.1 Makefile

The provided `Makefile` is basic and simple. It contains one variable `CFLAGS` in which any flags required for compilation may be passed to the compiler. Lines in a make file have the following format:

```
target : dependancies ; how to build it ;
```

Hence for hip_main where the binary is shipmain the following exists in the `Makefile`:

```
shipmain : hip_main.o utils.o ihip.o hd_notes.o hg_notes.o hp_notes.o
ihdnotes.o ihgnotes.o ihpnotes.o shipmain.c ; $(CC) $(CFLAGS) -o shipmain
shipmain.c hip_main.o utils.o ihip.o hd_notes.o hg_notes.o hp_notes.o
ihdnotes.o ihgnotes.o ihpnotes.o ;
```

Typing `make` without any arguments causes it to take the default target which is `all`. This contains the names of all the binaries as follows:

```
all: shipdm ssolar shipva shipva1 shipva2 shiprgc shipi shipdmc shipdmo shipdmv
shipdmx shipdmg shipj shipepe shipep stycmain stycep shipmain sdmsao shgnotes
shpnotes shdnotes shpauth
```

The target `clean` is also defined which removes all the binaries and '.o' files from the directory.

6.2 Writing your own code

To make use any of the routines provided ensure the necessary '.o' files are linked to the new code during compilation. A simple way to do this is to copy a similar line in the Makefile and modify it so it contains the name of the new source file.

6.2.1 pos_prop

The code for pos_prop (see Volume 1 of the printed catalogue) is also included in the src directory but is not in the Makefile. To use this code define a main loop which calls pos_prop and compile and link it with the new code. For example if myposprop.c is created the following would compile it:

```
cc -o myposprop myposprop.c pos_prop.c
```

Myposprop.c must contain the line:

```
#include "pos_prop.h"
```

7. Unix Utilities

The output of the above programs are ideal for use with the standard unix utilities such as `egrep, cut, join` and `nawk`. These may also be used to query the data files directly although this is not very efficient. For example, the following stores all HIP identifiers of entries in hip_main with a DSS chart in file hip.DSS:

```
cut -f2,70 -d "|" hip_main.dat | egrep D| cut -f1 -d"|"  >hip.DSS
```

On a sparc 20 this pipeline took over 5 minutes.

The resulting file may be used for further operations. For example to get a verbose print of each of these HIP entries the following could be typed:

```
shipmain 'cat hip.DSS'
```

However only some fields may be required. Because it is a verbose print each field is labelled so they may be selected by name using `egrep`. Hence to get `H1` plus all other fields with identifier in the description use:

```
shipmain 'cat hip.DSS' | egrep -i "H1 |identifier" > hipG.ids
```

This pipeline took under 10 seconds to run on the same machine.

These commands are very flexible. To use them properly consult the corresponding man pages. Perl is another scripting language which combines all of these features but is not installed as standard on all systems.

7.1 cut

The unix cut command is equivalent to the relational project operation. It allows one to vertically select columns from a file. So in the above example 'cut -f2,70' retrieves the second and 70th column from hip_main.dat. Note these are H1 and H69: cut does not know about column names, and it only works on column number. Hence for tyc_main and hip_main to get Tn or Hn cut -f(n+1) since tyc_main and hip_main start at field 0. The default field separator for cut is a tab, while the data files provided use '|'. Hence when using cut the option '`-d "|"`' must be used.

7.2 egrep/grep

Grep and egrep are similar, the main difference being that egrep allows more complex expressions. These functions provide functionality similar to the relation select operation. They search each line of a file to see if it matches the provided regular expression. Hence in the first example above 'egrep D' will match any line with the letter 'D' in it. Since these functions are not aware of field boundaries, this must be built into the expression.

'|' is a special character for egrep; hence it must be 'escaped' in expressions (it is the 'or' function otherwise, and will give unexpected results). Likewise '.' is a special character which matches any character in an expression. So, for example, to get entries with parallax > 9 and Johnson V magnitude = 4.nn an expression such as the following is required:

```
cut -f2,6,12 -d"|" hip_main.dat | \
egrep "^ *[0-9]+\| 4...\| *( 9|[1-9][0-9])"
```

Consider this expression in detail. First of all the cut gets the fields of interest i.e. H1 H5 and H11. Next the grep expression:

^	matches the beginning of sentence ($ matches the end of sentence).
*	matches zero or more occurrences of the character occurring before it: in this case any number of spaces.
[0-9]	The '[]' matches one character within the brackets. Number and character ranges may be specified, or individual characters or numbers. This example matches any single digit 0 to 9.
+	is like * but expects at least one occurrence of the character which appears before it.
|	to match the '|' field delimiter character it must be 'escaped' like this.
4	match a space followed by the digit 4.
...	match any 3 characters.
()	closing an expression in braces allows a sub expression to be constructed. Basically this expression says match a 9 or any number with two or more digits starting with the digit 1 through 9.
|	the 'or' operator.

Hence this says: start at the beginning of the line, skip any spaces then look for a number (any number) followed by the character '|', then look for a 4 followed by any 3 characters followed by '|'. Next skip any spaces and look for a number greater than 9. If grep at any time fails one of these tests then the line does not match the expression and will not be printed.

7.3 nawk

Nawk is a powerful context based script language. For example, to do something similar to the above nawk could be used as follows:

```
nawk 'BEGIN {FS="|"} \
$6~" +4..." {  if ($12 >= 9) {print $2"|"$4"|"$5"|"$6"|"$12 }}' hip_main.dat
```

`BEGIN` is a special tag which means this statement is only executed before reading the input. `FS` is the field separator.

The second statement says only for lines where $6 is 4.nn then if $12 is also at least 9, print columns 2 4 5 6 and 12.

7.4 perl

One problem with the above unix utilities is that it is often necessary to write scripts using all of the tools in complex pipe lines. The perl scripting language combines all of the

above features in a very fast language. For example to again perform the above selection the perl program would be as follows:

```
#!/usr/local/bin/perl -w

$infile="hip_main.dat";
#Open infput file
open(INFILE,"<$infile") || die ("Could not open $infile.");
# loop through file
while (<INFILE>)
{
   #Split line up using seperator
   @tline=split(/\|/);
   printf "%s|%s|%s|%s|%s\n",$tline[1],$tline[3],$tline[4],
                          $tline[5],$tline[11]
   if ($tline[5] =~ / +4.../ && $tline[11] >= 9)
}
close(INFILE);
```

The `split` operator allows a line of input to be split into fields according to a field separator which is a regular expression. Note that arrays start at reference zero hence to access H5 of the above line `$tline[5]` is used. Since there is no ';' after the `printf` statement it is only executed if the following if statement is true. In the if statements both numerical and regular expression operators have been used to provide the same match as in the previous examples.

With perl there is also lots of flexibility to do other interesting tasks, the following script computes the summary information given on certain fields in Section2.2 of Volume 1.

```
#!/usr/local/bin/perl -w
#Find number of occurences of each possible value in a given
#set of fields in the tyc_main file.
$infile="tyc_main.dat";

#Open infput file
open(INFILE,"<$infile") || die ("Could not open $infile.");
$RECS=0;
#The field numbers of the fields we want to do a statistic on
@FIELDS=(2,6,7,10,36,39,40,42,47,48,49,50,57);

##############################################
# loop through file
# Look at FIELDS count values accordingly
while (<INFILE>)
{
     # Dump "\r\n" from line
     chop;
     chop;

     #Split line up using seperator
     @tline=split(/\|/);
     $RECS +=1;

     #Look at each field we are interested in
     foreach $field (@FIELDS)
     {
        #Contruct Key name to be fieldname_value for the associative array
        $NAME=sprintf("T%s_%s",$field,$tline[$field]);
        #Increment this variable by 1 in the associative array %sums
        $sums{$NAME}++;
     }

}
close(INFILE);
#Print the results
&printInfo;

sub printInfo
{
   printf "Done %6d recs at TYC %s\n", $RECS, $tline[1];
   foreach $key (sort keys %sums)
   {
      printf "%8s     %6d\n",$key,$sums{$key};
   }
}
```

8. General Information

8.1 Using the search programs

The search programs are written such that they are assumed to be running in the cats directory, or at least in the same directory as the data file(s) to be searched. Hence to run any of the programs the user must change directory to be in the directory where the data files are to be found.

8.2 Quick Start (UNIX)

Copy the src directory from DISK1 to a directory on your hard disk (e.g. hipsrc). Change directory to the new directory and type make, for example:

```
cd ~/hipsrc
make
```

Modify the PATH variable in the '.cshrc' to include hipsrc. A line such as:

```
setenv PATH ${PATH}:~/hipsrc/src
```

will be necessary. This may also be typed on the command line but will then only have effect in the current shell and will be lost at logout.

For this change to take effect on a unix system:

```
source ~/.cshrc
rehash
```

When the compilation is complete, insert the required ASCII CD and change to the cats directory of the CD. On unix, the CD must be mounted with a command like:

```
mount /dev/cdrom /cdrom
cd /cdrom/cats
```

The system may have automounter running, in which case when the CD is placed in the drive it may appear under /cdrom/XXX. Otherwise it is necessary to have root privilege to perform mounts.

It should now be possible to run the compiled programs, for example (to search tyc_main for 1-13-1:

```
stycmain 1-13-1 | more
```

If the path was not modified or if this does not work, try:

```
../src/stycmain 1-13-1 | more
```

8.3 Possibility of copying data to disk

It is intended that once the routines have been compiled on the local system that they be used directly with the data files on the CD. They will function however equally well (and faster) if the data is transferred to disk. The data and index files may be copied using a normal cp command.

8.4 PC and MAC Users

Since all data files are written with PC end-of-line delimiter, they are viewable with normal PC applications. In principle, it should be possible to load the datafiles in MS-Access or Excel. Likewise MAC applications should be supported.

The search programs are written without PC end-of-line characters. However they should still compile using a PC ANSI/C compiler. The code must be compiled with the memory model huge, to achieve this it will be necessary to add a flag to the `CFLAGS` variable in the Makefile. The compiler documentation must be consulted to discover this flag (`-AH` for Microsoft C/C++). For editing it will possibly be necessary to convert the files using an application such as `unix2dos` although some of the work bench editors will allow editing without conversion.

On the MAC the apple file exchanger will do this conversion, or it may be done automatically when the files are copied, depending on how the system is set up. MAC programmers will have to modify the main loop to get a window and will probably have to find an alternative way for feeding identifiers to the program.

There may be a problem with the path specification in the `search_` routine in the following source files:

```
dmsa_o.c
hd_notes.c
hg_notes.c
hp_auth.c
hp_notes.c
```

A similar problem may occur with the `find_idx_` routine in these files:

```
ihdnotes.c
ihgnotes.c
ihpauth.c
ihpnotes.c
```

Simply edit the above files and where '`../notes`' is specified replace with '`..\notes`', to account for the different path specification syntax of DOS.

8.5 Modification of data files

The provided code works with the data files as they are defined in Volume 1. Any modification i.e. using `dos2unix`, will cause the record size to change which will cause the provided code to crash.

8.6 Location of src

All source code is provided on DISK1 in the `src` directory along with this readme.pdf and readme.doc).

William O'Mullane (CARA) May 14, 1997

Astrophysics Division SA, ESTEC, Noordwijk, Netherlands.
womullan@estec.esa.nl