A Set of C Utility Programs for Processing JPL Ephemeris Data

The Jet Propulsion Laboratory (JPL) has created a series of high fidelity ephemerides for the planets, and has made them available on their FTP site (`navigator.jpl.nasa.gov`). These ephemerides come in the form of a set of data files that come in both ASCII and binary formats.

The purpose of this document is to describe the design and use of a set of utility programs for working with the ASCII ephemeris data files created by JPL. These programs convert ASCII ephemeris data files to binary data files, concatenate these binary files, extract segments from them, display their contents, and interpolate quantities like position, velocity, libration angles, and nutation angles from the Tchebyshev coefficients they contain.

This document is split into two parts: a user's guide and a programmer's guide. The user's guide explains how to install and use the programs in this software distribution, while the programmer's guide provides design information to aid users who would either like to add functionality to these programs or to create an equivalent capability in languages other than C.

User's Guide

The information in this user's guide assumes that you will install and use this software on a computer running the UNIX operating system, because the programs in this software distribution were developed and tested on a machine running UNIX, specifically a Sun workstation running Solaris 2.6, and have not been ported to any other operating system. They were written in ANSI C, however, so they should run on any computer system that supports ANSI C.

Only one file in this distribution may not be portable to other systems, and that is the `Makefile`. While `make` itself has been ported to other platforms, this particular script calls the UNIX `sed` editor to modify one of the source files prior to compilation. Unfortunately, stream editors like `sed` are pretty much confined to the UNIX world. Therefore, some users may have to edit one file by hand in order to compile these programs on non–UNIX systems (fortunately it is only a one line change).

Installation. The installation of these programs is a three step process:

load the source files onto your system, compile the programs, and save the executables somewhere.

The first step in the installation process is the easiest. All of the C source code files need to be loaded into the same directory in order to be compiled, and each directory in the system is as good as any other.

The second step in the installation process, compiling the source code, is made easy by the UNIX `make` command. At the time that these programs were developed, the only two ephemerides available for download from JPL's FTP site were DE200 and DE405. The `Makefile` included with this distribution has options for both of these ephemerides (this `Makefile` is found in the same directory as the source code).

Suppose that you are interested in compiling the programs to work with the DE200 ephemeris. Then you would take the following steps to compile the source code:*

```
$ make 200
$ make all
```

The first invocation of `make` edits the file `ephem_types.h` to change the array size for the structure that holds the Tchebyshev coefficients, because the size of this array must be known at compile time. The second invocation of make actually compiles all of the programs in the distribution. The result of this will be that the source directory will now include a set of object files identified by a `.o` suffix (these can be deleted when finished with the compilation) and a set of executable program files.

This process must be repeated in order to make use of another ephemeris. Thus, to recompile these programs to work with the DE405 ephemeris, the first command would be repeated with the name of the ephemeris changed, *i.e.*, `make 405`, and then the second command, *i.e.*, `make all`, would follow.

As for the final step in the installation process, only a few rules constrain the location of the executable files. In particular, the programs `gdcon`, `jdcon`, and `rdeph` must be located in the same directory as `read.tcl` and the binary ephemeris data file. All of the other programs take the names of files to be opened as command line arguments, which can include an absolute or relative directory path. This means that the ultimate location of these programs is arbitrary, although placing them in the same directory as the data files is the most convenient choice.

Usage. For reasons that will be explained later, it is currently impossible

* Note that the `$` here represents the system prompt; the text that follows it is what you must enter.

to write C programs to reliably process the binary ephemeris files found on JPL's FTP server. Since binary files are more compact than ASCII files, this software distribution includes one program, `convert`, that converts ASCII files downloaded from JPL's FTP server into binary data files. All of the other programs in the distribution process the data contained in these data files (the format of these files is almost identical to that of JPL's binary ephemeris files; the differences will be discussed below).

Two of the programs in this distribution, `append` and `extract`, were created because the JPL ephemerides are broken up into a series of ASCII data files. These two programs, when used together, make it possible for users to create a single binary ephemeris file that spans an arbitrary time interval. To fully understand the use of these programs, you need to have some understanding of how JPL's (ASCII) ephemeris data files are organized.

The JPL program that generates the ephemerides apparently generates a stream of ASCII output that is broken up into a series of individual data files. One of them is called the header file and is named `header.###`, where `###` is the last three digits from the ephemeris name, *e.g.*, 200 for DE200. It contains information on the time span covered by the ephemeris, the math model coefficients used for the generation of the ephemeris, and—most important of all—information on how the data is organized in the remaining ephemeris data files. The last line of this header file consists of the words `GROUP 1070`, which signifies the remainder of the output stream is the ephemeris data proper.

The quantities of interest to ephemeris users, the positions and velocities of the planets or the nutation and libration angles, are computed for a given instant in time by a Tchebyshev series. Now, an entire ephemeris typically spans several centuries, and would require a prohibitively long Tchebyshev series to cover the entire span of the ephemeris. So the ephemeris is broken up into a series of intervals—typically 32 days in length—and a Tchebyshev series is generated for each interval. The coefficients for each of these series, together with the start and stop times of the interval, is stored in the ephemeris file as an individual record (which is essentially an array of floating point numbers.). A single file containing all of these coefficient records would be quite large, so they have been stored in a series of files that cover intervals of 20 years.

The ASCII data files found on JPL's FTP site are named according to the convention `asc[pn]****.###`. Here `[pn]` signifies a choice between `p` or `n` for the single character between '`c`' and the first '`*`', with `p` representing positive AD years and `n` representing negative BC years. The `****` represents the first year covered by the ephemeris (which starts around the first of the calendar year, but not necessarily on January first, though it always includes

that date), and the ### represents the ephemeris identifier using the same convention as the header files.

The first step in the creation of a binary ephemeris file is the conversion of one or several ASCII ephemeris files to binary ephemeris files. This conversion is performed by the program `convert`, which stores the data contained in an ASCII header file and one ASCII ephemeris file in a binary data file. It requires three command line arguments, which must be provided in the following order: the name of the ASCII header file, the name of the ASCII ephemeris file, and the name of the binary file that will hold the binary output. Therefore, to convert the ASCII ephemeris file `ascp2000.200` to a binary ephemeris file (named `binEphem.200` for the sake of argument), you would invoke `convert` as follows:

```
$ convert header.200 ascp2000.200 binEphem.200
```

When you execute `convert`, you will notice that it prints out status data while it executes. This information is designed to assure nervous users that everything is proceeding normally. The only times that you need to worry is when the `records rejected` count is not zero or when it prints out a warning message (warning messages are only printed when a fatal error is encountered; if this happens it means that `convert` has shut down prematurely). An example of `convert`'s output will be found in the example data conversion session shown below.

Unfortunately, the binary files created by `convert` are not identical to those found on JPL's FTP site. So long as `append` and `extract` are compiled by the same compiler, and run on the same type of computer, as `convert`, they will be able to read the files produced by `convert`. But the binary files created by `convert` cannot be ported to other computers reliably. The reason for this will be discussed in the programmer's guide below.

The program `append` appends the Tchebyshev coefficient records from one file to the end of another file (this means that the header information on the combined file comes from the first file). It requires two command line arguments, both of which are the names of binary data files. Therefore, to append the Tchebyshev coefficient records from `binEphem2.200` to those of `binEphem1.200` you would execute the command:

```
$ append binEphem1.200 binEphem2.200
```

The program `extract` extracts a set of Tchebyshev coefficient records that spans a user–supplied time interval. It takes two command line arguments; the first is the name of the file from which records are to be extracted, and

the second is the name of an empty file which is to receive this data*. The program prompts the user for the start and stop names. The example data conversion session that comes next will provide an example of its use.

The use of these programs can best be illustrated by an example. Suppose that you have downloaded the ASCII files `header.200`, `ascp1980.200`, and `ascp2000.200` and you want to create a binary data file called `binEphem.200` that covers the interval years 1990 through 2010 (inclusive). Then this is what you would do:

```
$ touch binEphem.200
$ touch binTemp1.200
$ touch binTemp2.200
$ convert header.200 ascp1980.200 binTemp1.200

   Writing record: 25
   Writing record: 50
   Writing record: 75
   Writing record: 100
   Writing record: 125
   Writing record: 150
   Writing record: 175
   Writing record: 200
   Writing record: 225

   Data Conversion Completed.

      Records Converted: 231
       Records Rejected:  0

$ convert header.200 ascp2000.200 binTemp2.200

   Writing record: 25
   Writing record: 50
   Writing record: 75
   Writing record: 100
   Writing record: 125
   Writing record: 150
   Writing record: 175
   Writing record: 200
   Writing record: 225
```

* None of the programs in this distribution create files. Therefore, all programs that write to a file must be given the name of an empty file. On UNIX systems, an empty file can be created by the `touch` command. On other systems, empty files can be created by other means such as saving an empty file with a text editor.

```
   Data Conversion Completed.

      Records Converted: 230
       Records Rejected:  0

$ append binTemp1.200 binTemp2.200
$ extract binTemp1.200 binEphem.200

   This program  extracts a segment from a binary ephemeris
   file. It requires start and stop times for this segment.
   These can  be input either as a pair of Julian  dates or
   as a Gregorian  date (in the form day, month, year) plus
   a Universal Time (in the form  hours, minutes, seconds).

   Input a Julian date (y/n) ?  n


   NOTE: Seconds is a real number (and thus requires
   a decimal point) while everthing else is an integer.

   Input Start Time:    Year ?  1990
                       Month ?  1
                         Day ?  1
                        Hour ?  0
                      Minute ?  0
                     Seconds ?  0.0


   Input Stop Time:     Year ?  2011
                       Month ?  1
                         Day ?  1
                        Hour ?  0
                      Minute ?  0
                     Seconds ?  0.0
$
```

At this point you would delete the temporary files, `binTemp1.200`, `binTemp2.200`, and the original ASCII files `header.200`, `ascp1980.200`, and `ascp2000.200` since they no longer serve any purpose.

The programs `print_header`, `read_record`, and `scan_records` were created to assist in the verification of the ASCII to binary data conversion programs. They simply read and display data from the converted binary data files so that it may be compared to the original ASCII data. These programs are included in this software distribution to assist those users who wish to modify the data conversion programs or who simply want to satisfy their curiosity about the data contained in the binary data files.

The program `print_header` extracts the header information that is stored in the binary file and prints it out in a format similar to the format of the JPL header files. It takes one command line argument: the name of the binary ephemeris data file. Therefore, to display the contents of the header records of the file `binEphem.200`, you would execute the command

```
$ print_header binEphem.200
```

The output of `print_header` will be sent to standard output, *i.e.*, it will scroll across the screen. On a UNIX system, this output can be saved to a file, called `listing` for this example, by the redirection operator

```
$ print_header binEphem.200 > listing
```

The program `read_record` prints out the contents of a binary array of double precision numbers. The first two of these numbers are times and the rest are the coefficients of the interpolating Tchebyshev polynomials. It takes two command line arguments. The first is the desired record number (starting with zero for the first data record), and the second is the name of a binary data file. Therefore, to view the contents of the second record of the binary file `binEphem.200` and save the results in a file called `listing`, you would execute the command

```
$ read_record 1 binEphem.200 > listing
```

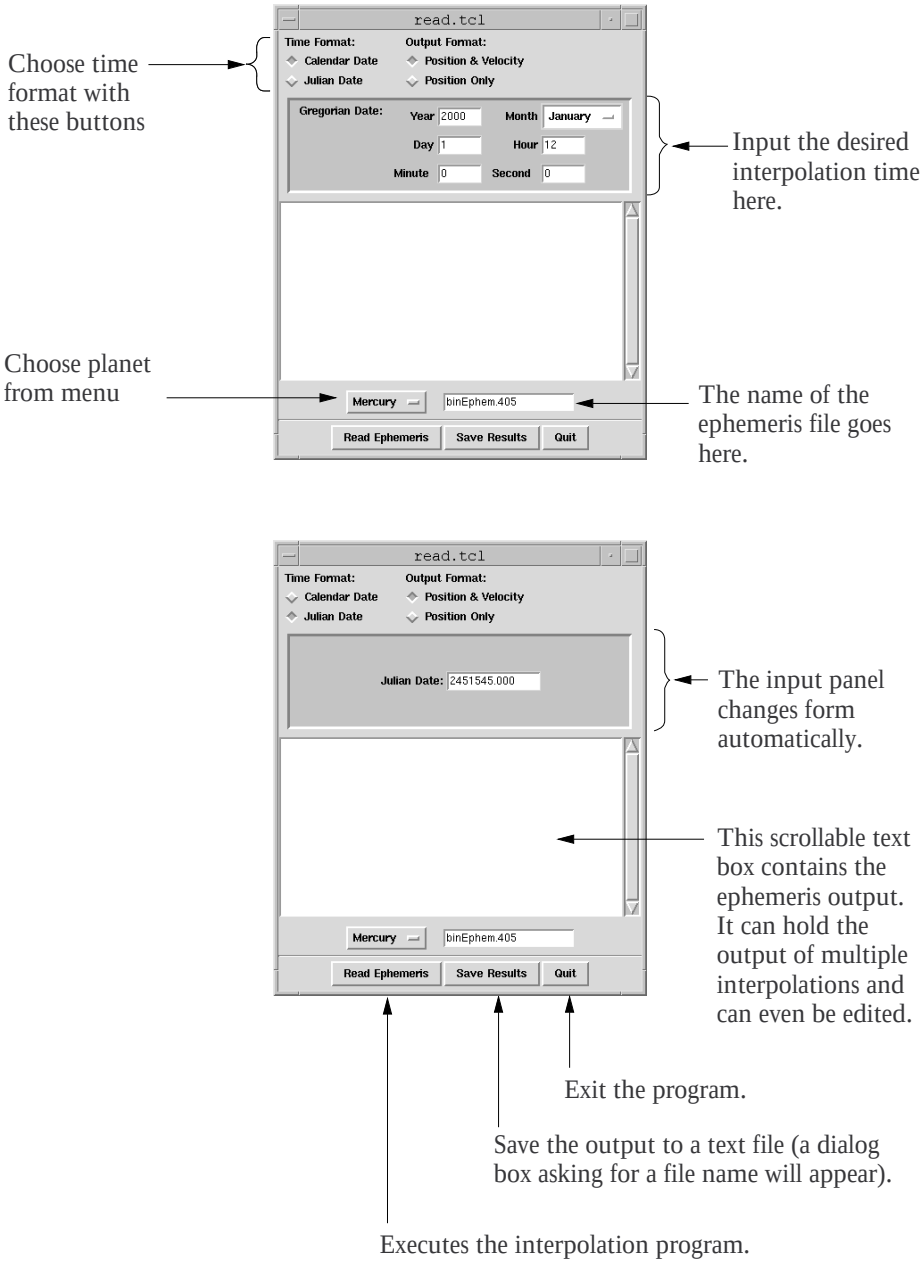
The program `scan_records` reads all of the records of interpolating polynomial coefficients in a binary data file and prints out the begin and end times for each of them. It is a useful tool for determining the time interval spanned by an ephemeris file and for searching for gaps or overlaps in files that have been joined. It takes one command line argument: the name of a binary ephemeris file. Therefore, to read the start and stop times of each record of Tchebeyshev coefficients in the file `binEphem.200` and save the results in a file called `listing`, you would execute the command

```
$ scan_records binEphem.200 > listing
```

The program `read.tcl` is a Tcl/Tk script that puts up a GUI dialog that allows user's to compute the position and velocity of a planet for a given time. This script only provides a user interface; the actual computations are performed by three C programs, `gdcon`, `jdcon`, and `rdeph`, that are executed by the Tcl command `exec`. It was created as an aid to debugging the ephemeris software and is included here as an example of how the ephemeris interpolation software could be put to use. An illustration of `read.tcl`'s GUI interface is shown in figure 1.

Normally, `read.tcl` is invoked from the command line in the usual way,

Figure 1: GUI interface for `read.tcl`



```
$ read.tcl
```

However, if you try this on a computer running the Common Desktop Environment (CDE), the GUI that you get will be messed up. This is because CDE messes up system wide resource files for its own benefit. To fix this problem, you need to use the **uncde** script, which is included in this distribution, as follows

```
$ uncde read.tcl
```

Calling the ephemeris. Many applications of the ephemeris data require that it be accessed from within a running program. Doing so is fairly simple, provided that the calling program is written in C. The outline of a C program that computes the the position and velocity of a planet would look something like this:

```
#include "ephem_read.h"
#ifndef TYPES_DEFINED
#include "ephem_types.h"
#endif

void main()
{
       char  ephemFileName[12];         /* Any size will work */
     double  Time;
        int  Planet;
  stateType  State;

  /*...other stuff (more declarations, get user input, etc)...*/

  Initialize_Ephemeris(ephemFileName);

  /*...still more stuff (maybe)...*/

  Interpolate_State( Time , Planet , &State );

  /*...do something with State (why else write the program?)...*/
}
```

The function `Initialize_Ephemeris` must be called once to initialize the ephemeris, then `Interpolate_State` or its siblings (see below) can be called. These functions can, of course, be called from any program written in a language that can access C functions.

Programmer's Guide

The software in this distribution is itself extensively commented, but no amount of source code documentation is ever sufficient to understand how a program works. This section will provide the supplemental information that will provide you with a complete understanding of how the programs in this distribution function. This information includes descriptions of the contents and format of the ASCII data files found on JPL's FTP site, the contents and format of the binary data files created by `convert`, the equations used for the Tchebyshev interpolation, and the general design of the ephemeris server used to perform user–requested interpolations.

ASCII header files. The header files contain general information about the ephemeris, such as the ephemeris name, time span, and model parameters, together with information about the organization of the Tchebyshev coefficients contained in the ephemeris data files. This last information is critically important: the Tchebyshev coefficient arrays cannot be read without it.

The header files are not large, and their general format can readily be seen by opening one with your favorite text editor. Upon opening one, you will see that, with the exception of two integer variables on the first line, all of the header file data is divided into groups numbered 1010, 1030, 1040, 1041, and 1050. These groups are separated by a blank line; a single line that contains the group number, `GROUP 1010` for example, and nothing else; and another blank line.

The first group, 1010, consists of three lines of ASCII text that names the ephemeris and identifies its begin and end times. The next group, 1030, repeats the start and stop times (in the form of Julian dates) and gives the time interval within which the Tchebyshev interpolation is done (in other words, the coefficients in each record in the data file are valid for this duration, which is typically 32 days). The next group, 1040, gives the names of various constants which are part of the least–squares fit that produced the ephemeris, while group 1041 gives their values. Note that the variables included in these groups can, and does, change from one ephemeris to the next. The last group, 1050, contains a table that completely describes the organization of the Tchebyshev coefficient arrays.

The dimensions of the arrays in the C source code are set, at compile time, according to the current value of the C macro `ARRAY_SIZE`, which is found at the top of the file `ephem_types.h` and is given its value by the `sed` editor when it is invoked by the `make` utility. The value that is assigned to `ARRAY_SIZE` is hard–coded into the `Makefile`, and its value must be the same as the value of `NCOEFF` in the header file of the ephemeris to be processed. `NCOEFF` is found on the first line of the header file.

The critical information in the header file, besides `NCOEFF`, consists of the third floating point number in `GROUP 1030`, which indicates the time span covered by each coefficient array; and the `GROUP 1050` table, which will be explained at length in the next section.

ASCII ephemeris files. The format of the ASCII ephemeris data files consists of a sequence of coefficient arrays, each of which is preceded by a line containing two integers. The first integer gives the record number for the coefficient data that follows, while the second gives the total number of records in the file. This line is followed by the values of the coefficient array, with three floating point values on each line. This pattern of integer pairs and floating point arrays is repeated until the end of the file.

The data in the coefficient array is arranged as follows: The first element gives the start time (as a Julian date) for the time interval spanned by the coefficients, while the second element (also a Julian date) gives the end time of the interval. Then come the Tchebyshev coefficients for the planets in the following order: Mercury, Venus, Earth–Moon barycenter, Mars, Jupiter, Saturn, Uranus, Neptune, Pluto, Moon, and Sun. These are followed by the Tchebyshev coefficients for a pair of nutation angles and, possibly, by coefficients for the three physical libration angles for the Moon (these are not present on all of the JPL ephemeris files).

The smaller and faster (shorter period) bodies in the ephemeris, most notably the Moon and Mercury, cannot accurately be represented by a Tchebyshev series that spans the typically 32–day interval used for the other ephemeris bodies. Therefore, their coefficient arrays are further subdivided into intervals called granules. The granule is, therefore, the fundamental time span covered by a particular set of Tchebyshev coefficients.

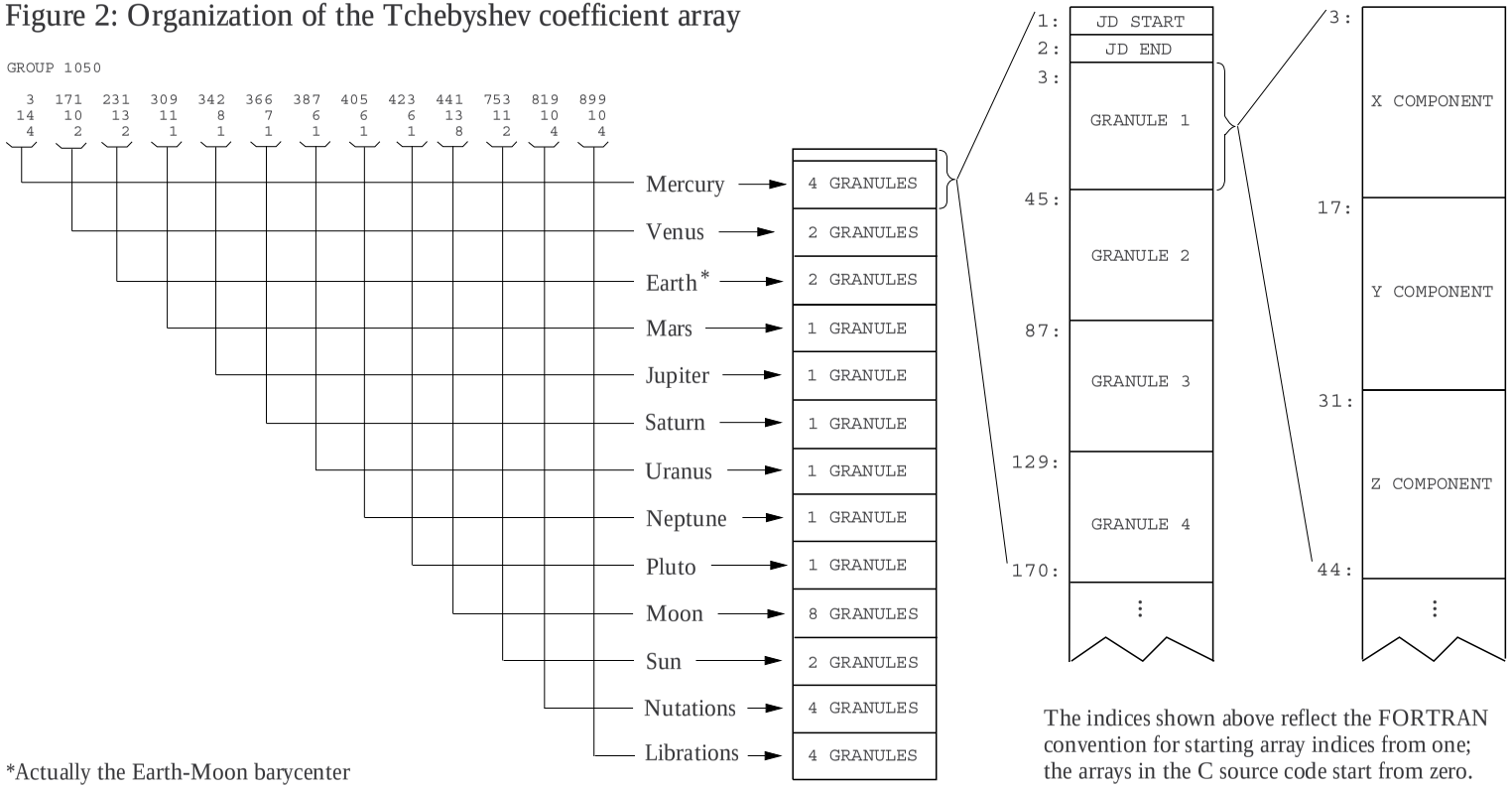
The same set of Tchebyshev coefficients are used to interpolate both the position and velocity*. They are arranged within each granule so that all of the coefficients for each vector are contiguous (this can be seen by using `read_record` to print out a record, and looking at the magnitudes of the coefficients). Figure 2 provides an illustration of this organization.

Each column in the `GROUP 1050` array represents either an ephemeris body (planet, Sun, or Moon), the nutation angles, or the lunar libration angles; these columns appear from left to right in the same order as the order of the coefficients in the array. The first row in the table gives the array location where the coefficients for the body (or angle set) begin, the second row gives the number of coefficients per component (or angle), and the third row gives the number of granules per time interval spanned by the array.

---

* For more details on this see: X. X. Newhall, *Numerical Representation of Planetary Ephemerides*, CELESTIAL MECHANICS, vol. 45, pp 305–310, 1989.

Figure 2: Organization of the Tchebyshev coefficient array



*Actually the Earth-Moon barycenter

The indices shown above reflect the FORTRAN convention for starting array indices from one; the arrays in the C source code start from zero.

A concrete example will illustrate how this information is used. Consider the first columns of the `GROUP 1050` table for the ephemeris `DE200`:

```
GROUP  1050

    3   147   183   ...
   12    12   ...   ...
    4     1   ...   ...
```

The first two entries in the coefficient array are the Julian dates for the beginning and the end of the interpolation interval, so the data for Mercury begins with the third element of the array. This data has 12 coefficients, one for each component of the position, and it is subdivided into 4 granules, so there are $3 \times 4 \times 12 = 144$ coefficients for Mercury in total. Add to this the two values of time at the beginning of the record, and you would find that the data for Venus starts with the 147^{th} element of the array. The starting location for the beginning of the coefficients for the Earth–Moon barycenter is left as an exercise (*hint:* the answer is given above).

Binary ephemeris files. The ASCII and binary ephemeris data files differ primarily in their encoding; the organization of the Tchebyshev coefficient arrays is identical in both types of files. The main difference between the ASCII and binary data files, aside from their encoding, is that the first two records of each binary data file contains all of the information that is found in the ASCII header file. This means that the functions that interpolate user–desired information can find all of the information they need from the binary files alone.

The header information is split between the two binary records because it cannot fit into only one of them. It does not fill both records, however, which means that each record also contains padding. This leads to a "record within a record" structure for the header records. In particular, each record is declared as a record plus a character array (the padding); then, each of these "subrecords" is declared. Therefore,

```
/* Declare header records */

struct headerOne {
   recOneType data;
   char pad[ARRAY_SIZE*sizeof(double) - sizeof(recOneType)];
   };

struct headerTwo {
   recTwoType data;
   char pad[ARRAY_SIZE*sizeof(double) - sizeof(recTwoType)];
```

```
   };

/* Declare data records */

struct recOneData {
       char  label[3][84];
       char  constName[400][6];
     double  timeData[3];
   long int  numConst;
     double  AU;
     double  EMRAT;
   long int  coeffPtr[12][3];
   long int  DENUM;
  long int  libratPtr[3];
  };

struct recTwoData {
   double constValue[400];
    };
```

These structures are made into data types for the sake of programming convenience:

```
typedef  struct headerOne  headOneType;
typedef  struct headerTwo  headTwoType;

typedef  struct recOneData  recOneType;
typedef  struct recTwoData  recTwoType;
```

Any program that reads or writes header data to a binary data file must first declare variables to hold that data. Unfortunately, C can't always resolve nested record components, which means that programs that access specific header variables must also declare variables to hold the subrecords. Therefore, the code that reads the ephemeris identifier from the header records would look like this:

```
/* Declare variables to hold header contents */

headOneType  H1;
headTwoType  H2;

/* Declare variables to hold subrecords */

recOneType  R1;
recTwoType  R2;

/* Declare a local variable */
```

```
long int Ephem;

/* Read a a value from the first header record */

R1    = H1.data;
Ephem = R1.DENUM;
```

The declaration of R2 is not strictly necessary in this example, since no value is extracted from it, but the declaration of H2 must be made if both header records are to be read into a program.

The general layout of the header data can be seen from the definitions of recOneType and recTwoType. The variable label contains the GROUP 1010 data, the array timeData contains the GROUP 1030 data, the entries in the constName array contain all of the names given in the GROUP 1040 data, the entries in the constValue array contain all of the names given in the GROUP 1041 data, and the array coeffPtr contains all of the GROUP 1050 data, except for the last column (lunar physical librations, which are found in the array libratPtr).

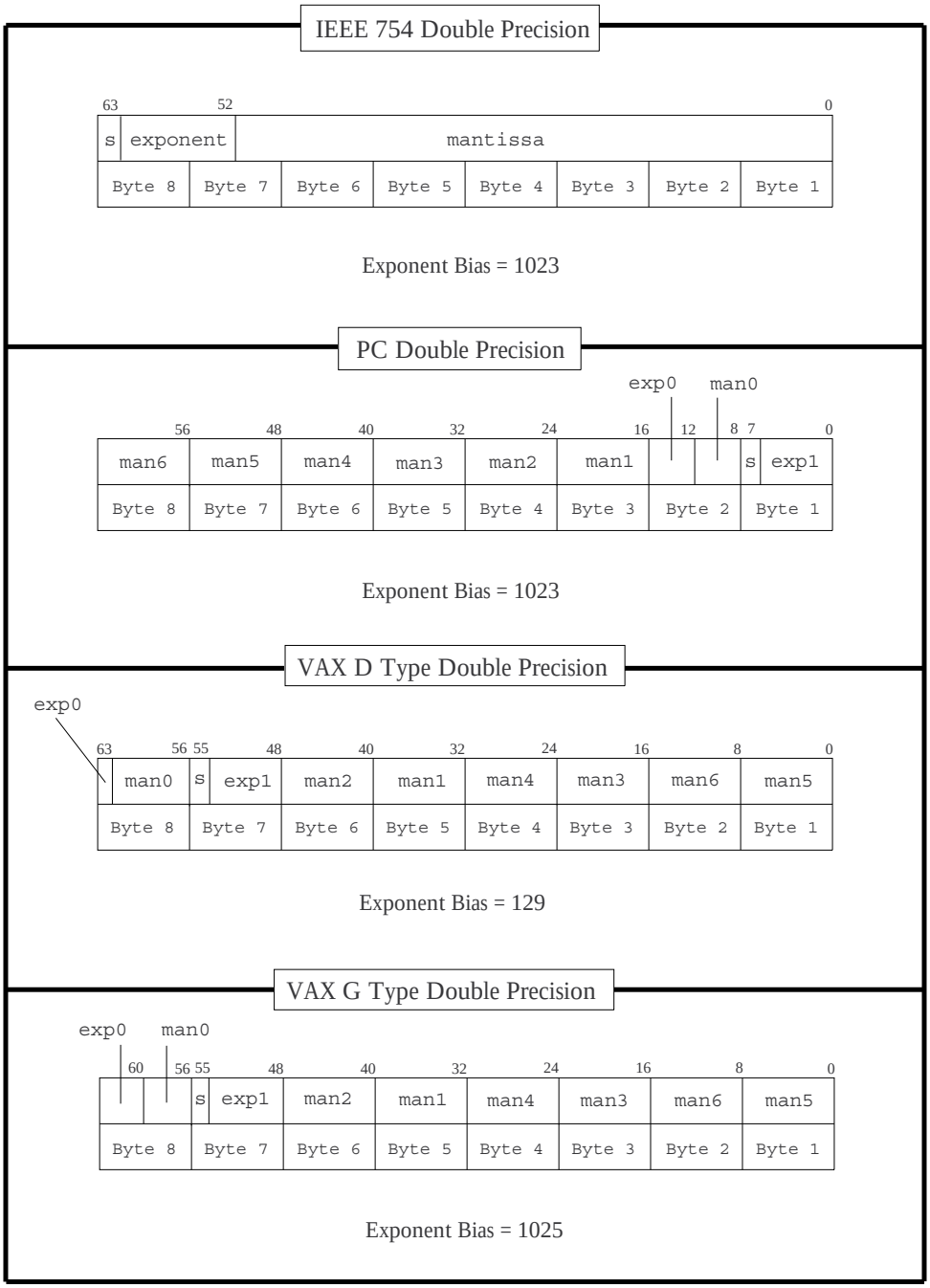
Note that the variables AU, EMRAT, and DENUM are somewhat redundant in that their values are contained in the GROUP 1041 data. The reason that these variables have been included in the record one data is that the format of these header records has been defined to match, as closely as possible, the definition of the data records of the FORTRAN software provided by JPL on their FTP site. Unfortunately, a perfect match of binary data formats is very difficult to achieve, for reasons that will be explained later.

The information that enables the interpolation functions to retrieve the coefficients for a particular body at a particular time comes from the GROUP 1050 table, which is stored in two variables: coeffPtr and libratPtr. The first array index of coeffPtr represents the columns of the table, and its second index represents the table's rows.

Binary file problems. While most modern microprocessors have registers and data busses of 32 bits, and will soon have 64 bit registers and busses, the byte remains a convenient unit of memory storage. ASCII characters are represented by a single byte, unicode characters are represented by two bytes, and numeric types are represented by some multiple of bytes. More importantly, computer memory is modeled as one long sequence of bytes by computer operating systems. This model presents two issues that affect the layout of binary files: byte ordering and memory alignment.

Byte ordering refers to the order in which the individual bytes of a multibyte word are stored in memory (or represented in CPU registers). The byte ordering of a given computer is determined by the CPU architecture, and

Figure 3: Common double precision formats



it comes in two flavors. A *little endian* machine, such as a PC or a DEC VAX, stores the least significant byte in the lowest order address (or the highest order byte of a register). A *big endian* machine, such as an Apple Macintosh or Sun workstation, stores the most significant byte in the lowest order address (or the highest order byte of a register). An illustration of these formats is presented in Figure 3. Note that the binary data files found on the JPL FTP server use the IEEE 754 format for double precision, which is the format used on Apple Macintoshes and Sun workstations.

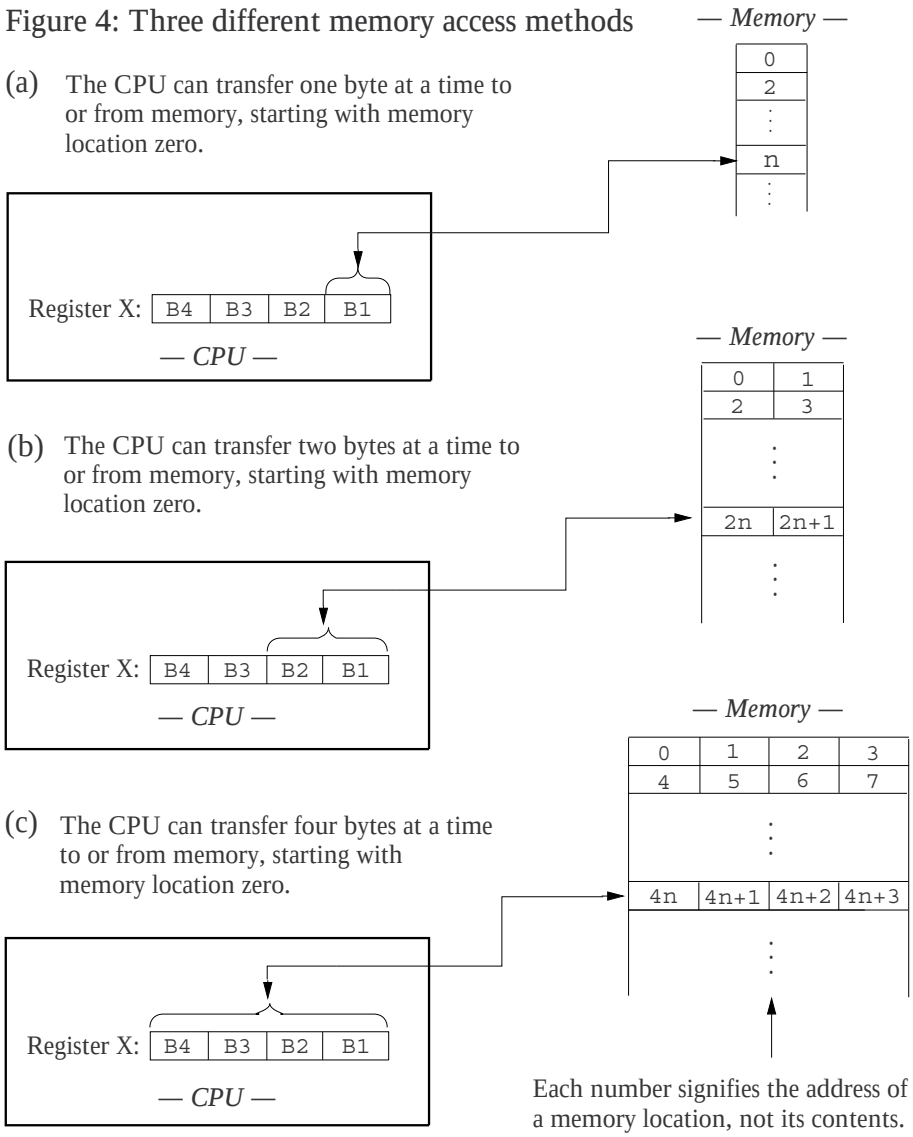
Converting a binary word from one ordering to another is a fairly simple task; the following C function converts a (32 bit) double precision number from big endian to little endian format:

```
/* First define a union globally... */

union doubleConvrt {
   double  F;
     char  C[8];
};

/* ...then the function itself */

doubleConvrt BE_to_LE (doubleConvrt X)
{
  doubleConvrt Y;

  Y.C[0] = X.C[7];
  Y.C[1] = X.C[6];
  Y.C[2] = X.C[5];
  Y.C[3] = X.C[4];
  Y.C[4] = X.C[3];
  Y.C[5] = X.C[2];
  Y.C[6] = X.C[1];
  Y.C[7] = X.C[0];

  return Y;
}
```

Clearly, byte ordering is not a serious problem. Memory alignment, on the other hand, is a major problem.

A CPU that has a 32 bit architecture will have the ability to transfer four bytes to or from memory at a time, but it may also be able to transfer individual bytes or pairs of bytes as well. However, the machine instructions that execute these operations typically impose constraints on memory addresses for the sake of efficiency.

Figure 4: Three different memory access methods



Consider, for example, the SPARC microprocessor found in Sun workstations. It has instructions to load a CPU register with one, two, or four bytes transfered from memory; these bytes are loaded into the least significant bits of the register and the higher order bits are cleared (except possibly for the sign bit). It also has instructions to transfer an entire register (four bytes), the least significant byte pair, or the least significant byte to memory. However, two byte transfers are only possible to (or from) memory addresses divisible by two, and four byte transfers are only possible to (or from) memory addresses divisible by four. These possibilities are illustrated by figure 4.

Suppose that a record structure made up of a three–element character array (requiring 8 bits of storage per character), a short (16 bit) integer, and a long (32 bit) integer is stored in memory as shown in figure 5. If this structure is packed into memory with no padding only 9 bytes of storage will be used. But this arrangement provides no convenient way to fetch either of the integers from memory because neither of them necessarily falls on a memory address divisible by two or four. Furthermore, fetching the entire structure requires nine single–byte read statements, some of which will have to be followed by byte shifting operations. This situation is an example of *unaligned memory access*.

The alternative to packed storage is to insert padding between the record components. This could be done in a number of ways, but one promising approach is shown in figure 5. Here 3 bytes of padding have been added so that the entire structure requires 12 bytes of storage, which means that it can be fetched from memory with only 3 four–byte read statements. In addition, either of the integers can be fetched with a single read statement (which does *not* need to be followed by a byte shifting operation). This situation is an example of *aligned memory access*.

So why is memory alignment a problem for programmers? This is the sort of thing that should be taken care of by the compiler, but that is the problem: some compilers insert padding (unused bits) so that they can generate machine code that uses aligned memory access and some do not. This is true for both C and FORTRAN (though C compilers seem to have a bias towards padding and FORTRAN compilers seem to have a bias against it). Furthermore, when padding is used, the resulting layout of data structures is highly dependent on the computer's architecture.

The binary ephemeris files created by `convert` are only affected by memory alignment because of the header records. The length of each header record is determined by how the compiler chooses to store the header records in memory. The data records are not affected because each array element uses the same amount of storage (eight bytes).

The C programming language does give programmers some control over the

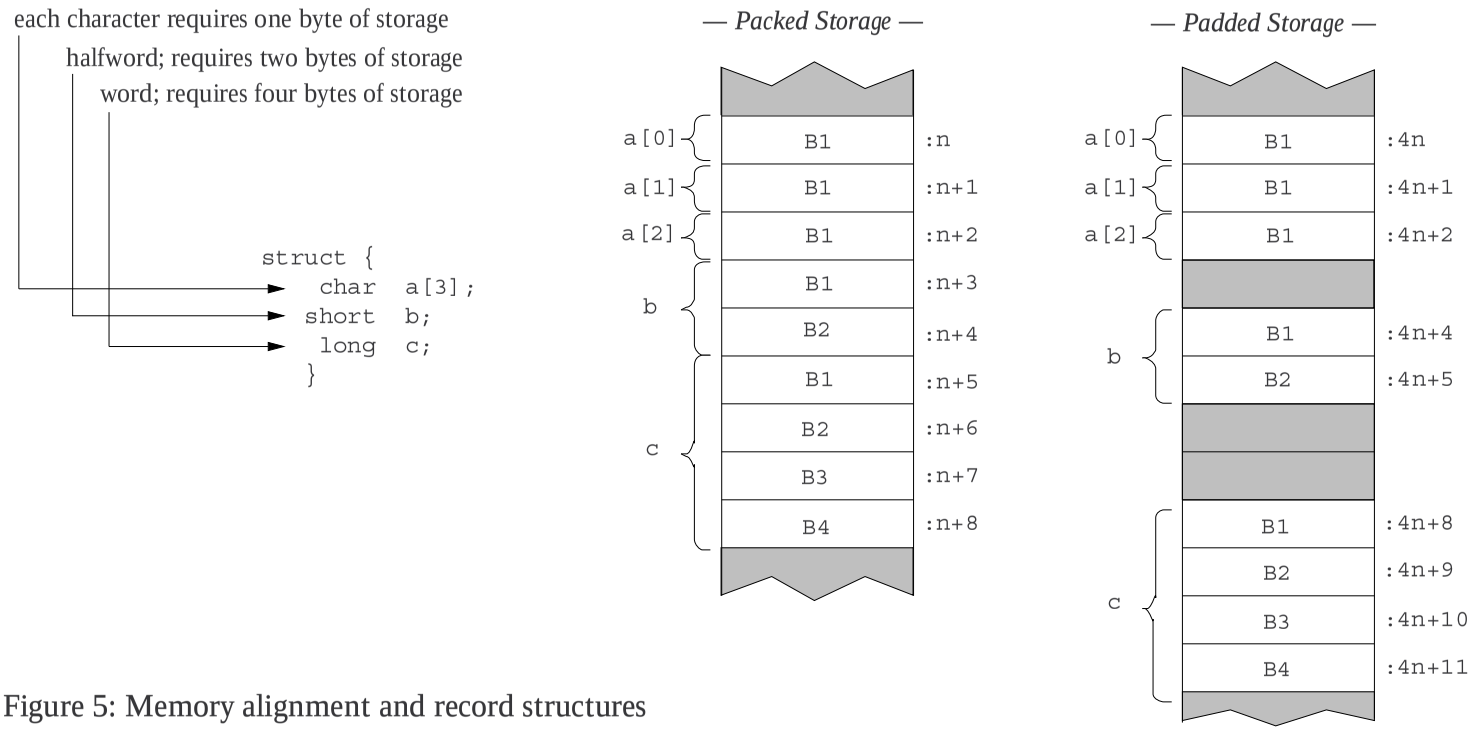


Figure 5: Memory alignment and record structures

layout of binary data structures, but it is troublesome to use and there is little point in taking the trouble unless an exact binary specification of the binary data files is available. Since such a specification does not exist for the binary files on the JPL FTP site, no attempt was made here to declare a particular format for the binary header records used by the programs in this software distribution.

Tchebyshev interpolation. Each component of the position or velocity of an ephemeris body at a given moment in time, as well as each libration or nutation angle, is given by a Tchebyshev series. These series all have the same form, so the interpolation of the x–component of the position vector can serve as an example for all these cases; its value at time t is given by

$$R_x(t) = \sum_{k=0}^{n} a_k \, T_k(t) \tag{1}$$

Here the functions $T_k(t)$ are Tchebyshev polynomials of the first kind, and the a_k are the coefficients whose values are stored in the ephemeris data files. The time, t, used in this equation is a *normalized* time, so that $-1 \leq t \leq 1$. This normalized time can be computed from the Julian date using the equation

$$t = \frac{T - T_0}{\Delta T} - 1$$

where T is the current Julian day number, T_0 is the Julian day number of the start of the coefficient granule, and ΔT is the interval spanned by the coefficient record.

Tchebyshev polynomials of the first kind are defined by the relation

$$T_k(t) = \cos(k \arccos t)$$

These functions may not look like polynomials, but by recursive application of the trigonometric identity

$$\cos nt = 2 \, \cos(n-1)t \cos t \, - \, \cos(n-2)t \tag{2}$$

it is possible to show that $\cos nt$ reduces to a polynomial in $\cos t$ of degree n. Thus, the $\cos(k \arccos t)$ in equation (2) reduces to a k^{th} order polynomial in $t \equiv \cos(\arccos t)$, so that the Tchebyshev polynomials are simply ordinary polynomials. The first four of them are

$$\begin{aligned} T_0(t) &= 1 \\ T_1(t) &= t \\ T_2(t) &= 2t^2 - 1 \\ T_3(t) &= 4t^3 - 3t \end{aligned}$$

and higher order ones can be computed from the general recursion relation

$$T_{k+1}(t) = 2t\,T_k(t) - T_{k-1}(t)$$

This recursion relation is used by the interpolation functions to compute the Tchebyshev polynomials that are used in equation (1), or the related expressions for the other components of position or any of the ephemeris angles.

The interpolation of velocity, on the other hand, is slightly more complicated. The basic interpolation equation is obtained by differentiating equation (1), so that

$$V_x(t) \equiv \dot{R}_x(t) = \sum_{k=0}^{n} a_k\,\dot{T}_k(t) \tag{3}$$

The derivatives of the Tchebyshev polynomials are given by the identity

$$\dot{T}_k(t) = k\,U_{k-1}(t) \tag{4}$$

where $U_k(t)$ is called a Tchebyshev polynomial of the second kind. These polynomials are defined by the relation

$$U_k(t) = \frac{\sin\left((k+1)\,\arccos t\,\right)}{\sin(\arccos t)}$$

and, like Tchebyshev polynomials of the first kind, they too are polynomials in t (though this is a little harder to show). The first four of them are:

$$\begin{aligned} U_0(t) &= 1 \\ U_1(t) &= 2t \\ U_2(t) &= 4t^2 - 1 \\ U_3(t) &= 8t^3 - 4t \end{aligned}$$

and, in general, they are defined by the recursion relation

$$U_{k+1}(t) = 2tU_k(t) - U_{k-1}(t)$$

Note, however, that equation (4) has a factor of k in each term, so that the recursion relation actually used in the interpolation function is

$$k\,U_{k-1}(t) = 2t(k-1)\,U_{k-2}(t) + 2\,T_{k-1}(t) - (n-2)\,U_{k-3}(t)$$

Finally, the value obtained by equation (3) must be adjusted by a scale factor. The value used for this scale factor in the interpolation software was taken from JPL's FORTRAN code. No mention of this value is made in any of JPL's documentation, though it obviously involves a change of units plus

an additional scale factor to give the coefficients of the velocity expansion the same magnitude as the position coefficients.

Source code organization. Each of the nine programs in this software distribution occupies one source file. The name of this source file is the name of the program followed by an attached suffix of '`.c`' (the one exception to this suffix rule is `read.tcl`; in this one case the suffix is not required and can be changed or dropped with no adverse consequences). These programs all depend on five additional source files.

The fundamental data structures common to all of the C functions and programs in this distribution are defined in the header file `ephem_types.h`. The file `ephem_util.c` contains a set of utility functions that are used by the ephemeris data conversion and display programs in this distribution. Its associated header file is `ephem_util.c`. The file `ephem_read.c` contains all of the functions used to process ephemeris data, and the prototypes of these functions are declared in the remaining source code file, `ephem_read.h`. These last two source files are most important to those who want to access ephemeris data from inside of their own programs, and an understanding of their design is the topic of the next section.

Design of the ephemeris server. The source file `ephem_read.c` creates a persistent ephemeris server, which can be thought of as a virtual machine that is active for the life of the program that uses it. This "machine" searches binary data files for Tchebyshev coefficient records and computes requested quantities from them. Most of its activities are hidden from programs that use it; to external programs the server looks like a set of five library routines: `Initialize_Ephemeris`, `Interpolate_Libration`, `Interpolate_Nutation`, `Interpolate_Position`, and `Interpolate_State`.

A schematic of the ephemeris server is shown in figure 6. It always maintains one Tchebyshev coefficient array in memory which always provides the coefficients used for interpolation. This array is loaded into memory by the function `Read_Coefficients`, which can only be called by one of the five user interface functions (because it is not visible to external programs).

The function `Initialize_Ephemeris`, which must be called once before any of the interpolation functions can be used, opens the binary ephemeris file, reads the header information from the first two records, then loads the first coefficient record in the file into memory. Once this is done, the server is ready to go to work.

The four interpolation functions have the same design. When any one of them is called, it first compares the requested interpolation time to the time span covered by the coefficient record currently loaded in memory. If the requested time is outside of this time span, the interpolating function

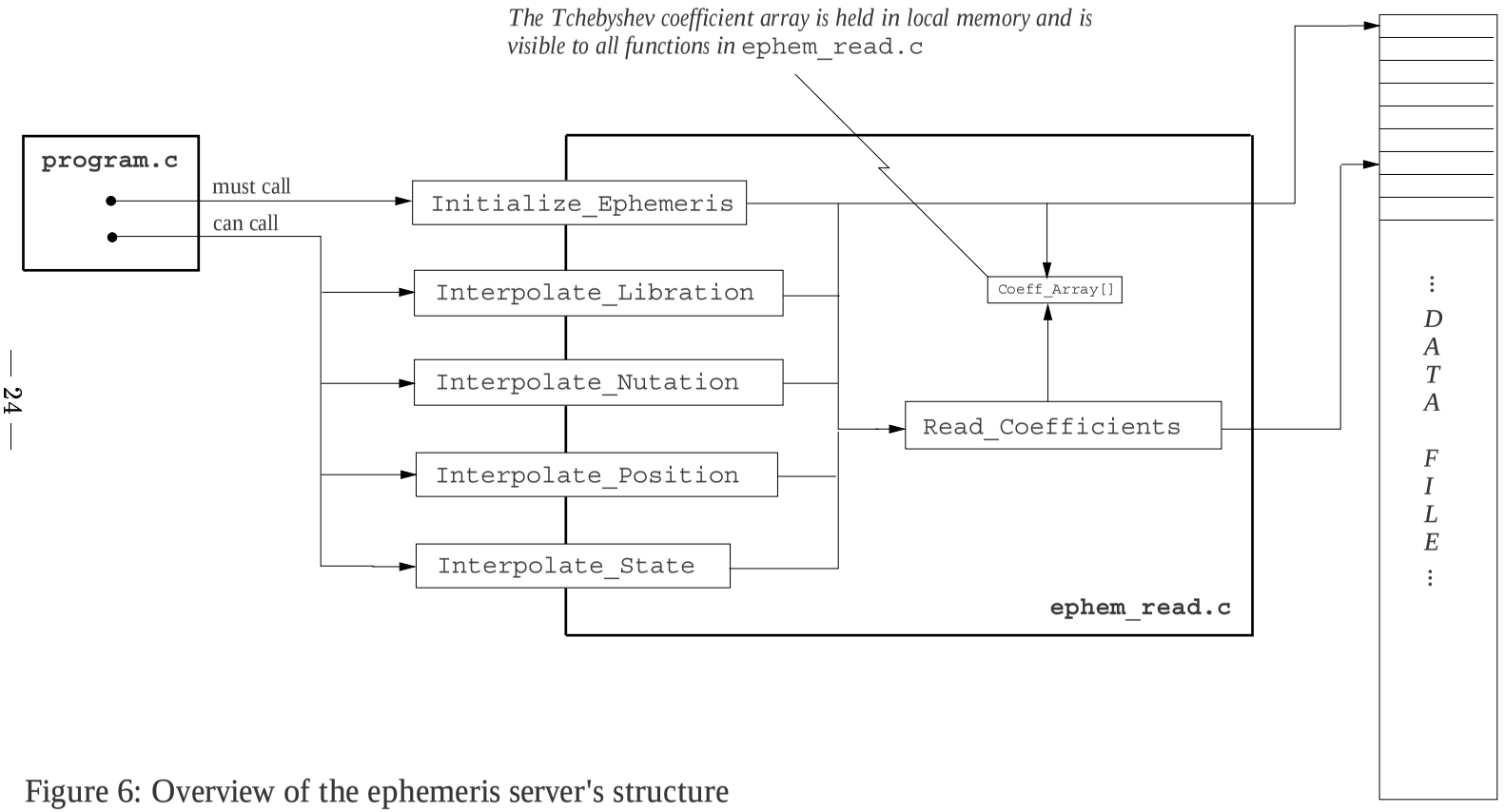


Figure 6: Overview of the ephemeris server's structure

calls `Read_Coefficients` which finds the record in the ephemeris file that covers the requested time and loads it into memory. Once the desired set of coefficients is loaded into memory, the interpolating function then computes the user–requested data.

Revision History

This document imports the text of the source code from the source code files themselves, so the listings given in the appendix always reflect the latest state of the code. Those who receive this software distribution and make significant modifications or additions to it are invited to change this document as well. The last element of any change to the documentation should be a paragraph added to this section which describes the changes made and gives the rationale for making them.

David Hoffman
August 3, 1998