KERNEL REQUIRED READING

Navigation Ancillary Information Facility

Prepared by
N. J. Bachman
W. L. Taber

NAIF Document No. 218.05
Last Modified: 06 OCT 1999

Table of Contents

Text Kernels

Revisions

October 5, 1999

This document differs from the previous version of March 25, 1992 in that it documents new features added to the kernel pool routines. The principal ones are:

- Kernel files containing string values are now supported. There is a new character fetch routine GCPOOL.
- Numeric kernel variables may be fetched as integers or double precision numbers via the new fetch routines GIPOOL and GDPOOL.

  The routines GDPOOL, and GIPOOL supersede the routine RTPOOL, which is now deprecated.
- There is a new routine DTPOOL which allows an application to determine whether a variable specified by name is present in the kernel pool, and if so, to retrieve the variable's dimension and data type.
- There are now three new routines PCPOOL, PDPOOL and PIPOOL which support direct insertion of data into the kernel pool without reading an external file.
- There is a new interface routine LMPOOL that allows SPICE programs to load text kernels directly from memory instead of requiring a text file.
- There is a new routine SZPOOL which returns kernel pool definition parameters.
- There is a new routine DVPOOL which allows the removal of a variable from the kernel pool.
- There is a new routine GNPOOL which allows an application to obtain the names of the variables present in the kernel pool.

This document now describes the kernel pool as an abstract data structure, rather than emphasizing the view of the kernel system as a reading mechanism for text kernels.

In addition some minor edits were performed to improve clarity.

Also, the quoting style was changed from British to American.

Introduction

This document describes the SPICELIB "kernel pool" system, which provides a robust, flexible way to load data into memory in a SPICE based program, either from SPICE text kernel files or via a subroutine interface.

A variety of SPICE text kernels that are read into and accessed through the kernel pool. These include:

- Text PCK kernels
- Leapseconds kernels
- SCLK kernels
- I kernels
- Frame kernels

Do not confuse SPICE text kernels with text "transfer" versions of SPK, CK, PCK or DAF files produced by the utilities SPACIT or TOXFR. The text transfer files are not intended to be used with the kernel pool and do not conform to the SPICE text kernel format.

The kernel pool system interface is composed of two parts: a text file format and kernel pool access software. The software includes routines that read files conforming to the format, routines that allow direct insertion of data into the pool via subroutine calls, routines that fetch data from the kernel pool, routines that return information about the current state of the pool, and utilities that manipulate various aspects of the pool.

The SPICE text kernel format has a "name = value" structure similar to the format used to assign values to variables in languages such as C and FORTRAN. Details of the format are described below.

The kernel pool may be viewed abstractly as a repository of associative arrays which map names to lists of numeric or string values. The kernel pool allows SPICELIB-based programs to read data from SPICE text kernel files while maintaining the "name = value" associations established in the files. Alternatively, associative arrays may be inserted into the kernel pool via the pool's programming interface.

Once name-value associations have been stored in the kernel pool, you may access the stored data through kernel pool look-up routines. These look-up routines use the names as keys to find the associated values. The look-up and other access routines are described in detail below.

The SPICE text kernel file format

As the name implies, SPICE text kernel files contain only ASCII text. An additional restriction on the contents of SPICE text kernel files is that they contain no non-printing characters, such as tabs or formfeeds.

We illustrate this format by way of example using an excerpt from a SPICE text planetary constants kernel (PCK) file. The format description given below applies to all SPICE text kernels; the specific data names shown below apply only to text PCK files.

```
Planets first. Each has quadratic expressions for the direction
(RA, Dec) of the north pole and the rotation of the prime meridian.
Planets with satellites (except Pluto) also have linear expressions
for the auxiliary (phase) angles used in the nutation and libration
expressions of their satellites.

\begindata

BODY399_POLE_RA        = (    0.       -0.64061614  -0.00008386  )
BODY399_POLE_DEC       = (  +90.       -0.55675303  +0.00011851  )
BODY399_PM             = (   10.21   +360.98562970  +0.          )
BODY399_LONG_AXIS      = (    0.                                 )

BODY3_NUT_PREC_ANGLES  = (  125.045     -1935.53
                            249.390     -3871.06
                            196.694   -475263.
                            176.630   +487269.65
                            358.219    -36000.    )

\begintext

Each satellite has similar quadratic expressions for the pole and
prime meridian. In addition, some satellites have nonzero nutation
and libration amplitudes. (The number of amplitudes matches the
number of auxiliary phase angles of the primary.)

\begindata

BODY301_POLE_RA      = (  270.000   -0.64061614  -0.00008386   )
BODY301_POLE_DEC     = (  +66.534   -0.55675303  +0.00011851   )
BODY301_PM           = (   38.314  +13.1763581    0.           )
BODY301_LONG_AXIS    = (    0.                                 )

BODY301_NUT_PREC_RA  = (  -3.878  -0.120  +0.070  -0.017   0.     )
BODY301_NUT_PREC_DEC = (  +1.543  +0.024  -0.028  +0.007   0.     )
BODY301_NUT_PREC_PM  = (  +3.558  +0.121  -0.064  +0.016  +0.025  )
```

```
\begintext

Finally, we include the radii of the satellites and planets.

\begindata

BODY399_RADII    = (     6378.140    6378.140     6356.755  )
BODY301_RADII    = (     1738.       1738.        1738.     )
```

In this example are several comment blocks. All are introduce by the control word:

```
\begintext
```

A comment block may contain any number of comment lines. Once a comment block has begun, no special characters are required to introduce subsequent lines of comments within that block. A comment block is terminated by the control word

```
\begindata
```

This control word also serves to introduce a block of data that will be stored in the kernel pool. Each of these control words must appear on a line by itself.

Each variable definition consists of three components:

1. A variable name.
2. An assignment directive. This must be "=" (direct assignment) or "+=" (incremental assignment).
3. A scalar or vector value.

Direct assignments supersede previous assignments, whereas incremental assignments are added to previous assignments. For example, the series of assignments

```
BODY301_NUT_PREC_RA  = -3.878
BODY301_NUT_PREC_RA += -0.120
BODY301_NUT_PREC_RA += +0.070
BODY301_NUT_PREC_RA += -0.017
BODY301_NUT_PREC_RA += 0.
```

has the same effect as the single assignment

```
BODY301_NUT_PREC_RA = (  -3.878  -0.120  +0.070  -0.017   0 )
```

Dates, e.g.,

```
FOOBAR_CALIBRATION_DATES = ( @31-JAN-1987,
                             @2/4/87,
                             @March-7-1987-3:10:39.221 )
```

may be entered in a wide variety of formats. There are two restrictions regarding the format of dates. They may not contain embedded blanks, and they must begin with the character

```
@
```

Internally, dates are converted to TDB seconds past J2000 as they are read. As a result, dates, are treated as numeric data in the pool.

Strings may be supplied by quoting the string value.

```
MISSION_UNITS = ( 'KILOMETERS',
                  'SECONDS',
                  'KILOMETERS/SECOND' )
```

If you need to include a quote in the string value, use the FORTRAN convention of "doubling" the quote.

```
MESSAGE = ( 'You can''t always get what you want.' )
```

The types of values assigned to a kernel pool variable must all be the same. If you attempt to make an assignment such as the one shown here:

```
ERROR_EXAMPLE = ( 1, 2, 'THREE', 4, 'FIVE' )
```

The kernel pool reader will regard the assignment as erroneous and reject it and any subsequent kernel pool assignments that appear in the text kernel.

Managing Kernels

The generic kernel loader FURNSH

For the SPICE system to use kernel files, the files must be made known to the system and opened at run time. This activity is called "loading" kernels. SPICELIB provides a simple subroutine interface for this purpose. The principal kernel loading subroutine is called FURNSH (pronounced "furnish"). The kernel system also provides a small set of routines that enable an application to find the names and attributes of kernels that have been loaded via FURNSH. These routines are all entry points of the subroutine KEEPER.

In earlier versions of SPICELIB, kernels were loaded via routines specific to various SPICELIB subsystems: SPK, CK, PCK, EK, kernel pool. The binary kernel systems also supported unloading kernels. All of the old loaders and unloaders are still provided in SPICELIB, but these routines should no longer be called directly. FURNSH should be called instead.

NAIF now recommends that instead of calling various kernel loaders, applications load kernels using a "metakernel." A metakernel is a SPICE text kernel that lists the names of the kernels to be loaded. At run time, the application supplies the name of the metakernel as an input argument to FURNSH. For example, instead of loading kernels using the code fragment

```
CALL LDPOOL ( 'leapseconds.ker'          )
CALL LDPOOL ( 'mgs.tsc'                  )
CALL SPKLEF ( 'generic.bsp',    HANDLE1 )
CALL CKLPF  ( 'mgs.bc',         HANDLE2 )
CALL PCKLOF ( 'earth.bpc',      HANDLE3 )
CALL EKLEF  ( 'mgs.bes',        HANDLE4 )
```

one now may write

```
CALL FURNSH ( 'kernels.txt' )
```

where the file kernels.txt is a SPICE text kernel containing the lines

```
\begindata

KERNELS_TO_LOAD = ( 'leapseconds.ker',
                    'mgs.tsc',
                    'generic.bsp',
                    'mgs.bc',
                    'earth.bpc',
                    'mgs.bes'          )
```

This technique has the advantage of enabling a user to change the set of kernels loaded by the application without modifying source code.

It also possible to use FURNSH to load kernels in the older SPICELIB style: the names of kernels to load can be supplied as input arguments to FURNSH.

For example, instead of using the series of loader calls shown earlier, one now may write

```
INTEGER               FILEN
PARAMETER           ( FILEN = 255 )

INTEGER               NKER
PARAMETER           ( NKER  = 5 )

INTEGER               I

CHARACTER*FILEN       KERNLS ( NKER )

DATA                  KERNLS / 'leapseconds.ker',
.                              'mgs.tsc',
.                              'generic.bsp',
.                              'earth.bpc',
.                              'mgs.bes'        /

DO I = 1, NKER
   CALL FURNSH ( KERNLS(I) )
END DO
```

Kernel priority

The older SPICELIB loaders allow users to prioritize kernel files via load order: kernels loaded later have higher priority than kernels loaded earlier. FURNSH follows the same convention. When kernels are listed in a metakernel, those appearing later in the list have higher priority. The old prioritization scheme also applies to kernels supplied directly as arguments to FURNSH.

Path symbols

Inside a metakernel, it is sometimes necessary to qualify file names with their pathnames. To reduce both typing and the need to continue file names over multiple lines, metakernels allow users to define symbols for paths. This is done using the kernel variables

```
PATH_NAMES
PATH_SYMBOLS
```

To create symbols for path names, one assigns an array of path names to the variable PATH_NAMES. Next, one assigns an array of corresponding symbol names to the variable PATH_SYMBOLS. The nth symbol in the second array represents the nth path name in the first.

Finally, one prefixes with path symbols the kernel names specified in the KERNELS_TO_LOAD variable. Each symbol is prefixed with a dollar sign to indicate that it is in fact a symbol.

Suppose in our example above that the MGS kernels reside in the path

```
/flight_projects/mgs/SPICE_kernels
```

and the other kernels reside in the path

```
/generic/SPICE_kernels
```

Then we can add paths to our metakernel as follows:

```
\begindata

PATH_NAMES   = ( '/flight_projects/mgs/SPICE_kernels',
                 '/generic/SPICE_kernels'             )

PATH_SYMBOLS = ( 'MGS',
                 'GEN' )


KERNELS_TO_LOAD = ( '$GEN/leapseconds.ker',
                    '$MGS/mgs.tsc',
                    '$GEN/generic.bsp',
                    '$MGS/mgs.bc',
                    '$GEN/earth.bpc',
                    '$MGS/mgs.bes'          )
```

It is not required that paths be abbreviated using path symbols; it's simply a convenience.

Note the symbols defined here are not related to the symbols supported by a host shell or any other operating system interface.

Finding out what's loaded

SPICELIB-based applications may need to determine at run time which files have been loaded. Applications may need to find the DAF or DAS handles of loaded binary kernels so that the kernels may be searched. Some applications may need to unload kernels to make room for others, or change the priority of loaded kernels at run time.

SPICELIB provides kernel access routines to support these needs. For every loaded kernel, an application can find the name of kernel, the kernel type (text or one of SPK, CK, PCK, or EK), the kernel's DAF or DAS handle if applicable, and the name of the metakernel used to load the kernel, again if applicable.

The routine KTOTAL returns the count of loaded kernels of a given type. The routine KDATA returns information on the nth kernel of a given type. The two routines are normally used together. Following is an example of how an application could retrieve summary information on the currently loaded SPK files:

```
      CALL KTOTAL ( 'SPK', COUNT )

      IF ( COUNT .EQ. 0 ) THEN
         WRITE (*,*) 'There are no SPK files loaded at this time.'
      ELSE
         WRITE (*,*) 'The loaded SPK files are: '
         WRITE (*,*)
      END IF

      DO WHICH = 1, COUNT

         CALL KDATA( WHICH,  'SPK',  FILE, FILTYP,
     .               HANDLE, SOURCE, FOUND        )
         WRITE (*,*) FILE

      END DO
```

Above, the input argument 'SPK' is a kernel type specifier. The allowed set of values is

```
SPK  --- All SPK files are counted in the total.
CK   --- All CK files are counted in the total.
PCK  --- All binary PCK files are counted in the
         total.
EK   --- All EK files are counted in the total.
TEXT --- All text kernels that are not meta-text
         kernels are included in the total.
META --- All meta-text kernels are counted in the
         total.
ALL  --- Every type of kernel is counted in the
         total.
```

In this example, FILTYP is a string indicating the type of kernel. HANDLE is the file handle if the file is a binary SPICE kernel. SOURCE is the name of the metakernel used to load the file, if applicable. FOUND indicates whether a file having the specified type and index was found.

SPICELIB also contains the routine KINFO which returns summary information about a file whose name is already known. KINFO is called as follows:

```
      CALL KINFO ( FILE, FILTYP, SOURCE, HANDLE, FOUND )
```

Unloading kernels

SPICELIB-based applications may need to remove loaded kernels. Possible reasons for this are:

- To make room to load other kernels.
- To change the priority of loaded kernel data.
- To change the set of kernel data visible to SPICELIB.

The routine UNLOAD "removes" kernels from a SPICE application. For binary kernels, the meaning of this is simple: the file is closed, and SPICELIB data structures referring to the file's contents are adjusted to reflect the absence of the file.

Text kernels and metakernels may be unloaded as well. Unloading a metakernel involves unloading the files referenced by the metakernel. Text kernels are unloaded by clearing the kernel pool and then reloading the other text kernels not designated for removal.

Note that unloading text kernels has the side effect of wiping out kernel variables that have been set via the kernel pool's subroutine write access interface. It is important to consider whether this side effect is acceptable when writing code that may unload text kernels or metakernels.

The routine used to unload kernels is UNLOAD. UNLOAD is called as follows:

```
CALL UNLOAD ( KERNEL )
```

Fetching data from the kernel pool

The values of variables stored in the kernel pool may be retrieved using the subroutines:

`GCPOOL`	Used to fetch character data from the kernel pool.
`GDPOOL`	Used to fetch double precision data from the kernel pool.
`GIPOOL`	Used to fetch integer data from the kernel pool. Note that internally, all numeric data is stored as double precision values. This interface is provided as a convenience so that users may retrieve integer data directly from the kernel pool without having worry about converting from double precision values to integers.

The calling sequences have the same appearance and meaning for all three routines.

```
CALL GCPOOL(NAME, FIRST, ROOM, NVALUES, VALUES, FOUND)
CALL GDPOOL(NAME, FIRST, ROOM, NVALUES, VALUES, FOUND)
CALL GIPOOL(NAME, FIRST, ROOM, NVALUES, VALUES, FOUND)
```

where

`NAME`	is the name of the item to retrieve
`FIRST`	is the index of the first item to retrieve from the array of values associated with NAME.
`ROOM`	is the number of values that may be stored in the output array VALUES.
`NVALUES`	is the number of items stored in VALUES
`VALUES`	is the output array of values associated with NAME. The data type of VALUES depends upon the routine: for GCPOOL, VALUES is an array of strings; for GDPOOL, VALUES is an array of double precision numbers, for GIPOOL, VALUES is an array of integers.
`FOUND`	indicates whether or not the requested data is is available in the kernel pool.

See the headers of these subroutines for a more extensive discussion of their arguments and use.

Informational routines

Four routines are provided for retrieving general information about the contents of the kernel pool.

`DTPOOL`	Returns information about the existence, dimension and type of kernel pool variables.
`EXPOOL`	Returns information on the existence of a kernel pool variable.
`GNPOOL`	Allows retrieval of names of kernel pool variables that match a string pattern.
`SZPOOL`	Returns information about the size of various structures used in the implementation of the kernel pool.

These routines are discussed at length in their respective headers.

Changing Kernel Pool Contents

The main way in which you change the contents of the kernel pool is by "loading" a SPICE text kernel with the routine FURNSH. However, the kernel pool also provides a several other routines that allow you to change the contents of the pool.

`CLPOOL`	clears (initializes) the kernel pool, deleting all the variables in the pool.
`LMPOOL`	Similar in effect loading a text kernel via FURNSH, but the text kernel is stored in an array of strings instead of an external file.
`PCPOOL`	Allows the insertion of a character variable directly into the kernel pool without supplying a text kernel.
`PDPOOL`	Allows the insertion of a double precision variable directly into the kernel pool without supplying a text kernel.
`PCPOOL`	Allows the insertion of an integer variable directly into the kernel pool without supplying a text kernel.
`DVPOOL`	allows deletion of a specific variable from the kernel pool. (CLPOOL deletes all variables from the kernel pool.)

The following code fragment shows how the data normally provided in a leapseconds kernel could be loaded via LMPOOL. See the headers of the other routines for specific details regarding their use.

Below, BUFFER is a character array and N is the size of the array.

```
INTEGER                LNSIZE
PARAMETER            ( LNSIZE = 80 )

CHARACTER*(LNSIZE)     TEXT ( 27 )

TEXT( 1) = 'DELTET/DELTA_T_A =   32.184'
TEXT( 2) = 'DELTET/K         =    1.657D-3'
TEXT( 3) = 'DELTET/EB        =    1.671D-2'
TEXT( 4) = 'DELTET/M = (6.239996D0 1.99096871D-7)'
TEXT( 5) = 'DELTET/DELTA_AT  = ( 10, @1972-JAN-1'
TEXT( 6) = '                      11, @1972-JUL-1'
TEXT( 7) = '                      12, @1973-JAN-1'
TEXT( 8) = '                      13, @1974-JAN-1'
TEXT( 9) = '                      14, @1975-JAN-1'
TEXT(10) = '                      15, @1976-JAN-1'
TEXT(11) = '                      16, @1977-JAN-1'
TEXT(12) = '                      17, @1978-JAN-1'
TEXT(13) = '                      18, @1979-JAN-1'
TEXT(14) = '                      19, @1980-JAN-1'
```

```
TEXT(15) = '                  20, @1981-JUL-1'
TEXT(16) = '                  21, @1982-JUL-1'
TEXT(17) = '                  22, @1983-JUL-1'
TEXT(18) = '                  23, @1985-JUL-1'
TEXT(19) = '                  24, @1988-JAN-1'
TEXT(20) = '                  25, @1990-JAN-1'
TEXT(21) = '                  26, @1991-JAN-1'
TEXT(22) = '                  27, @1992-JUL-1'
TEXT(23) = '                  28, @1993-JUL-1'
TEXT(24) = '                  29, @1994-JUL-1'
TEXT(25) = '                  30, @1996-JAN-1'
TEXT(26) = '                  31, @1997-JUL-1'
TEXT(27) = '                  32, @1999-JAN-1)'

CALL LMPOOL ( TEXT, 27 )
```

Detecting Changes in the Kernel Pool.

Since loading SPICE text kernels tends to happen only at program initialization, a routine that relies on data in the kernel pool may run more efficiently if it can store a local copy of the values needed and update these only when a change occurs in the kernel pool. Two routines are available that allow a quick test to see whether kernel pool variables have been updated.

`SWPOOL` Sets up a "watcher" on a variable so that various "agents" can be notified when a variable has been updated.

`CVPOOL` Indicates whether or not an agent's variable has been updated since the last time an agent checked with the pool.

See the headers of these routines for details and examples of their use.

Saving the contents of the kernel pool

If you need to capture a persistent copy of the contents of the kernel pool. Use the routine WRPOOL.

SPICE subsystems that rely on SPICE text kernels.

PCK-related routines

The PCK kernel is SPICELIB's source of the planetary constants needed to define the size, shape, and orientation of planets and satellites. The PCK text file format and routines which access PCK data are described in the PCK Required Reading.

Time conversion routines

Routines that retrieve leapseconds or SCLK data directly from the kernel pool are documented in the TIME and SCLK Required reading files, respectively.

Frame transformation routines

See the FRAMES Required Reading for a discussion of frame definition kernels.

Summary of Routines

Each kernel pool subroutine name consists of a mnemonic which translates into a short description of the routine's purpose.

Many of the routines listed below are entry points to another subroutine. If they are, the parent routine's name will be listed inside brackets preceding the mnemonic translation.

```
BODFND         ( Find values from the kernel pool  )
BODVAR         ( Return values from the kernel pool )
CLPOOL [POOL]  ( Clear the pool of kernel variables )
CVPOOL [POOL]  ( Check variable in the pool for update )
DTPOOL [POOL]  ( Data for a kernel pool variable )
DVPOOL [POOL]  ( Delete a variable from the kernel pool )
EXPOOL [POOL]  ( Confirm the existence of a pool kernel variable )
FURNSH [KEEPER]( Furnish a program with SPICE kernels )
GCPOOL [POOL]  ( Get character data from the kernel pool )
GDPOOL [POOL]  ( Get d.p. values from the kernel pool )
GIPOOL [POOL]  ( Get integers from the kernel pool )
GNPOOL [POOL]  ( Get names of kernel pool variables )
KDATA  [KEEPER]( Kernel Data )
KINFO  [KEEPER]( Kernel Information )
KTOTAL [KEEPER]( Kernel Totals )
LDPOOL [POOL]  ( Load variables from a kernel file into the pool )
LMPOOL [POOL]  ( Load variables from memory into the pool )
PCPOOL [POOL]  ( Put character strings into the kernel pool )
PDPOOL [POOL]  ( Put d.p.'s into the kernel pool )
```

```
PIPOOL [POOL]  ( Put integers into the kernel pool )
STPOOL [POOL]  ( String from pool )
SWPOOL [POOL]  ( Set watch on a pool variable )
SZPOOL [POOL]  ( Get size limitations of the kernel pool)
UNLOAD [KEEPER]( Unload a kernel )
```

Summary of Calling Sequences

```
BODFND ( BODY,   ITEM )
BODVAR ( BODY,   ITEM,    DIM, VALUES )
CLPOOL ()
CVPOOL ( AGENT,  UPDATE )
DTPOOL ( NAME,   FOUND,   N,   TYPE )
DVPOOL ( NAME )
EXPOOL ( NAME,   FOUND )
FURNSH ( FILE )
GCPOOL ( NAME,   START,   ROOM,   N,      CVALS,  FOUND )
GDPOOL ( NAME,   START,   ROOM,   N,      DVALS,  FOUND )
GIPOOL ( NAME,   START,   ROOM,   N,      IVALS,  FOUND )
GNPOOL ( NAME,   START,   ROOM,   N,      KVARS,  FOUND )
KDATA  ( WHICH,  KIND,    FILTYP, SOURCE, HANDLE, FOUND )
KINFO  ( FILE,   FILTYP,  SOURCE, HANDLE, FOUND )
KTOTAL ( KIND,   COUNT )
LDPOOL ( NAME )
LMPOOL ( CVALS,  N )
PCPOOL ( NAME,   N,      CVALS )
PDPOOL ( NAME,   N,      DVALS )
PIPOOL ( NAME,   N,      IVALS )
STPOOL ( ITEM,   NTH,    CONTIN,  STRING,  SIZE,   FOUND )
SWPOOL ( AGENT,  NNAMES, NAMES )
SZPOOL ( NAME,   N,      FOUND )
UNLOAD ( FILE )
```