May 31, 1990

INTEROFFICE MEMORANDUM

TO: DISTRIBUTION

FROM: S. Collins - CMSS

SUBJ ORIGINAL DOCUMENT RELEASE - SIS NAV-135

Attached is the signed original release of subject document.

Please contact me at 30965 if you have any questions.

PROJECT MAGELLAN
SOFTWARE INTERFACE SPECIFICATION
Cover Sheet

NUMBER: NAV - 135
REVISION: Original
DATE: 4/25/90

SIS NAME: NAIF SP-KERNEL

DOMAIN:

System	Subsystem	Program	Make/Use
MOS	NAV	MGNSPK	MAKE
MOS	TPS	MHR	USE
MOS	IDPS	ALT_DAILY	USE (ON EDR)
MOS	SCIENCE	N/A	USE (ON EDR)
MOS	SDPS	CONTROL/ESP	USE (ON EDR)
MOS	RES	RAS	USE (ON EDR)

Generating Computer System: SUN

PURPOSE OF INTERFACE (SUMMARY): Specifies the S/C and planetary ephemeris file on the SCI EDR (TPS - 101).

SIS REFERNECE: SFOC-2-DPS-CDB-EPHEMERIS

INTERFACE MEDIUM

Disk File: [ ]

Magnetic Tape: [X] Tracks: 9 Density: 6250 (SCI EDR)

Other: [X] SFOC Local Area Network (TPS)

SIS COORDINATOR: J. Ekelund

SIGNATURES:

Approvals	Name		Date
GDS Manager	J. Gunn	Jody Gunn	5/23/90
S/W System Eng.	R. Halvorstad	R. E. Halvorstadt	5-23-90

(continued)

Subsystem/ Prog. Set	Position	Name		Date
NAV	Sys Eng.	J. Ekelund	J. Ekelund	2/1/90
TPS	Sys Eng.	E. Wilson	J. Dunn for E.A.W	5/15/90
IDPS	Sys Eng.	P. Jepsen	~~for PLJ~~	5/15/90
SDPS	Sys Eng.	M. Jin	M. Jin	5/15/90
Science Manager		T. Thompson	S/ Wall for TMT	5/21/90
SFOC System Eng.		J. Holladay	Jaya Holladay	5/23/90
RES/RAS	Sys Eng.	K. Wong	K. Wong	2/6/90

Distribution List:

K. Andersen
S. Collins
J. Ekelund
J. Gunn
R. Halverstadt
P. Jepsen
M. Jin
J. Miller
T. Thompson
E. Wilson
K. Wong

Document Change Log

DATE	VERSION	Change Authorization
4/25/90	Original	Initial

SFOC Software Interface Specification (SIS) Module	Name: SFOC-2-DPS-CDB-Ephemeris Publication Date: 04-25-90 Final

MODULE TITLE: Spacecraft and Planet Ephemerides

PURPOSE: Module describes information needed by users of Navigation Ancillary Information Facility (NAIF) S and P Kernel (SPK) file. SPK files serve to deliver spacecraft, planet, and satellite ephemeris data from a flight Project Navigation Team to the SFOC Central Data Base for subsequent distribution to users.

DATA FLOW:

From	To	Data Object	Physical Medium
DPS	CDB	Text files	FTP data transport software

PREPARER(S): C. Acton, I. Underwood Cha IMU

Baseline System Engineer CDB Subsystems	R. Borgen	5/10/90
Baseline System Engineer MHR Subsystem	M. Tankenson	5/10/90
MGN TLM Subsystems Engineer	for B. Wilson	5/14/90
Ground Data System Office Manager	J. Gunn	5/23/90
SFOC System Engineer SFDU Control Authority	J. Johnson	5/11/90
Technical Group Supervisor	J. Wilson	5/10/90
Technical Group Supervisor	C. Carr	5/10/90
SFOC System Engineer	G. Dawson	Gil Dawson 5/24/90
SFOC System Engineer	J. Holladay	5/23/90

DOCUMENT LOG

Date	Page No.	Status
12-09-87	All	Draft
01-08-88	All	Prelim.
01-09-90	All	Final

Date	Page No.	Status

LIST OF TBD ITEMS

Page	Section	Item

DISTRIBUTION

Acton, C.	301-125
Borgen, R	301-350
Carr, C.	301-350
Dawson, G.	264-728
FPSO Library	T-1607
Greenberg, E.	301-280R
~~Gunn, J.~~	~~230-201BG~~
Holladay, J.	264-728
Jo4hnson, J.	264-728
Tankenson, M.	301-260
Underwood, I.	301-125
~~Wilson, B.~~	~~170-206~~
Wilson, J.	301-260

For additional copies of this document, contact the FPSO Library at X3-0606, or T-1607

CONTENTS

SECTION 1

GENERAL DESCRIPTION

1.1 PURPOSE OF DOCUMENT

This Software Interface Specification module describes information needed by users of Navigation Ancillary Information Facility (NAIF) S and P kernel (SPK) files. SPK files are the principal means by which ephemeris data for spacecraft, planets, satellites, comets, and asteroids are delivered from a flight project Navigation Team (NAV) to the Space Flight Operations Center (SFOC) Central Data Base (CDB) for subsequent distribution to user teams or organizations.

1.2 SCOPE

This SIS module departs somewhat from the normal structure in that the detailed object description is contained in three separate NAIF documents which are included in this SIS as appendices. Appendices A and B describe the structure of the multimission SPK file and the underlying Double Precision Array File (DAF) architecture. In addition, they describe those portions of NAIF's portable FORTRAN 77 toolkit which provide users easy access to the ephemeris data contained within SPK files. Appendix C describes an SPK utility program, SPACIT, which provides text-to-binary conversion, binary-to-text conversion, and file summary capabilities for SPK files.

This SIS module applies to the orbital phase of the Magellan mission.

1.3 APPLICABLE DOCUMENTS

1. SCDA 0007-00-04 — Common Data Access Software Specification Document, May 13, 1988.

2. DDN Protocol Handbook: Internet Protocol, RFC791, 9/81, Internet Control Message Protocol, RFC791, 9/81; User Datagram Protocol, RFC768, 8/80; Transmission Control Protocol, RFC1981, September 1981.

3. JJPL0006-01-00 — JPL SFDU Usage and Description, Issue No. 5; edited by J. Johnson; March 24, 1988. JPL D-5325.

4. SFPC0038-XX-08 — Space Flight Operations Center, Software Interface Specifications, February 7, 1990.

   a. SFOC0038-01-06-02 — SFOC-1-MHR-Mgn-SCIEDR, SAR and Altimeter EDR/TEDR Tapes, November 28, 1988.

b.	SFOC0038-01-03-01	SFOC-1-DPS-MGN-Ephemeris, Spacecraft Ephemeris File, Final, May 27, 1988.
c.	SFOC0038-00-08	SFOC-1-CDB-MGN-TimesFile, Key S/C Event Times File, Preliminary, February 26, 1988.
d.	SFOC0038-01-06-02	SFOC-1-MHR-MGN-SCIEDR, Revision C, November 28, 1988.

1.4 FUNCTIONAL DESCRIPTION

SPK files are the *physical* realization of two *logical* portions of the SPICE kernel system -- the S-kernel (spacecraft ephemeris) and the ephemeris portion of the P-kernel (planet, satellite, comet, and asteroid ephemerides). An SPK file will yield state vectors that match those obtained from the source NAV files with differences smaller than the inherent accuracy of the original data.

1.4.1 Data Source, Destination, and Transfer Method

For Magellan Project purposes, SPK files are normally created by the NAIF program MGNSPK on a workstation assigned to the NAV Team. The files are labeled and packaged in accordance with the CCSDS Version 1 SFDU methodology.

SPK files can also be created by anyone with access to the NAIF SPICELIB toolkit library. However, that software does not include the SFDU-formatting capability built into MGNSPK.

Data used in creating a Magellan SPK file are extracted from three sources:

1. A Spacecraft Ephemeris File provided by the Navigation Subsystem
2. A Planetary Ephemeris File provided by the Navigation Subsystem
3. A Key S/C Event Times File provided by the Sequence Generation Subsystem

All three files are produced on the JPL UNISYS computers. The two ephemeris files are produced in binary NAVIO format. They are converted to text format, moved to the NAV workstation (using FTP), and reconverted to binary NAVIO format as described in Applicable Document #4b. The Key Times File is produced in the text format described in Applicable Document #4c. It is also transferred to the NAV workstation using FTP.

One copy of each SFDU created by MGNSPK is transferred to the SFOC CDB at a node and directory specified by the operator of the program, for example, cd27:/u/cdb/mgn/input/ephem. The transfer is effected using the CDB FTS service.

A second copy of each SFDU is transferred to the Magellan Data Management Team (DMAT). The transfer is effected using industry-standard 1/2 inch 6250 b.p.i. magnetic tape.

1.4.2 Pertinent Relationships with Other Interfaces

The SFOC CDB Data Management System reads catalog information from the SFDU labels in order to make appropriate entries in the CDB catalog. However, it does not read the actual SPK ephemeris data.

The Magellan High Rate System (MHR), in simply copying a text format SPK file onto an EDR tape, is not directly concerned with the SFDU structure or contents, or with the format or contents of the actual SPK file. MHR obtains all the information needed about a particular SPK file from the CDB catalog.

Members of Magellan science teams receive SPK files on MHR EDR/TEDR tapes. The science teams are the end users of SPK files, reading them in order to compute state vectors. The science teams may read the catalog information in the SFDU labels at their discretion.

Changes to SIS module SFOC-1-DPS-MGN-EPHEMERIS, which incorporates Magellan SIS No. NAV-124 and Magellan SIS No. NAV-104, or to SIS module SFOC-1-CDB-MGN-TimesFile, which incorporates Magellan SIS No. SGS-108, could affect the contents of this module.

1.4.3 Identification and Labeling

1.4.3.1 SPK File Identification

SPK files delivered to the CDB will adhere to the following file-naming convention:

MGNnnnnn.TSP

Here, "nnnnn" is simply a sequential, monotonically increasing number starting at 00001. (There will likely be some SPK files produced by the Magellan NAV Team which will *not* get delivered to the SFOC CDB for one reason or another, so the values of "nnnnn" will probably not be sequential within the CDB.)

Encoding Orbit Number and/or Upload Number in the file name is not attempted for two reasons. First, while the nominal Magellan operations plan is to have each file span only one orbit, there is no equivalent software restriction. Second, it seems reasonable to expect that redeliveries of SPK files to the CDB will occasionally be required due to operations problems; this would cause a file name conflict unless some alternate approach was adopted.

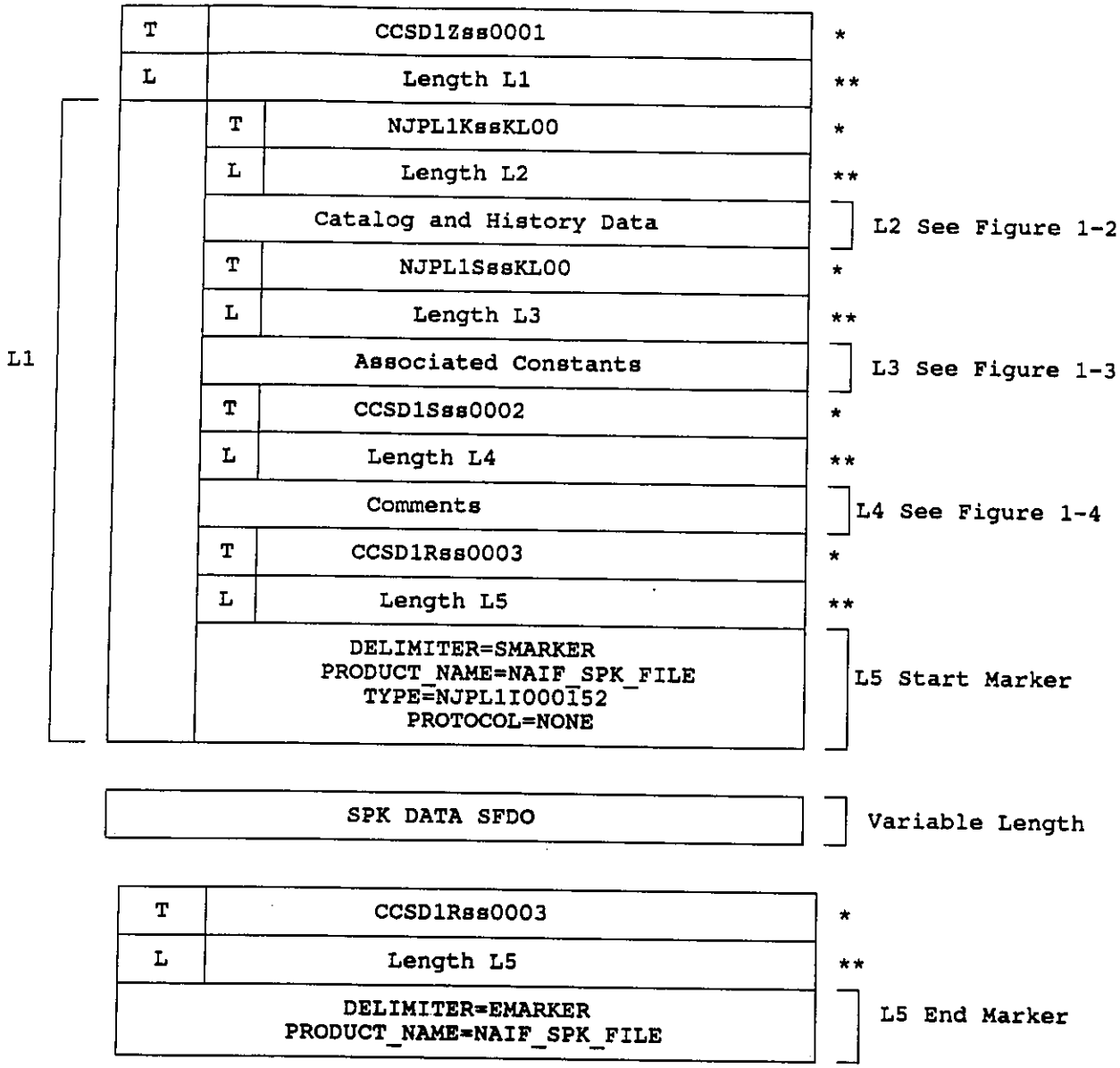
Information about Orbit Number(s) and Upload Number(s) covered by a given file are available within the SFDU Catalog and History label section.

1.4.3.2 SPK File Labeling

SPK files delivered to the SFOC CDB are structured as Standard Format Data Units (SFDU) per Magellan-era SFDU standards as stated in Applicable Document #3.

In applying the SFDU concept to SPK files, the "marker" approach is employed, wherein the total length of an SPK file is *not* required in a header record. A keyword=value type data object is used for label records.

The structure of the SPK SFDU is depicted in Figure 1-1.



* "ss" = Spares set to ASCII "00"

** For version ID=1 SFDU- the type used by the Magellan Project - the "Length" field is an eight-character ASCII integer. See Applicable Document #3, JPL SFDU Usage and Description.

Figure 1-1 SPK SFDU Structure

DATA_OBJECT_TYPE=EPHEMERIS	
PRODUCT_VERSION_TYPE=type	Note 1
DATA_SET_NAME=MGNyyyyy.TSP	
MISSION_ID=4	
MISSION_NAME=MAGELLAN	
SPACECRAFT_NAME=MAGELLAN	
SPACECRAFT_ID=18	Note 2
PROCESS_TIME=YYYY-MM-DDThh:mm:ss:fff	
VERSION_ID=01	
UPLOAD_ID=cnnnnc	
ORBIT_NUMBER=nnnnn	
OBJECT=ORBIT_TIME_RANGE_GROUP TIME_RANGE_NUMBER=1 ORBIT_START_TIME=YYYY-MM-DDThh:mm:ss:fff ORBIT_STOP_TIME=YYYY-MM-DDThh:mm:ss:fff END_OBJECT=ORBIT_TIME_RANGE_GROUP	Note 3
OBJECT=MAPPING_TIME_RANGE_GROUP TIME_RANGE_NUMBER=1 MAPPING_START_TIME=YYYY-MM-DDThh:mm:ss:fff MAPPING_STOP_TIME=YYYY-MM-DDThh:mm:ss:fff END_OBJECT=MAPPING_TIME_RANGE_GROUP	
EFFECTIVE_TIME=YYYY-MM-DDThh:mm:ss:fff	
NAV_UNIQUE_ID="32*C"	Note 4

NOTES:
1. "type" will be filled in. Currently acceptable values are: PREDICT and ACTUAL.
2. Spacecraft number (assigned by DSN) is 18 for Magellan.
   CAUTION: SPICELIB subroutines use -18.
3. The MGNSPK program produces a separate SPK file for each orbit, hence, only one obit start/stop pair and mapping start/stop pair will appear in this SFDU, and the value for TIME_RANGE_NUMBER will always be 1.
4. Up to 32 characters, left justified and blank filled, including blanks, enclosed in double quotes.
   Legend: c = alphanumeric
   n = digit
   yyyyy = sequential numbers starting at 00001

Figure 1-2 Catalog and History Data

This SFDU includes three distinct label sections in addition to the actual SPK ephemeris data. The label sections are:

1. CATALOG AND HISTORY DATA

2. ASSOCIATED CONSTANTS

3. COMMENTS

These sections are depicted in Figures 1-2, 1-3, and 1-4. Note that for the Magellan Project the ASSOCIATED CONSTANTS and COMMENTS sections are currently unused (left vacant).

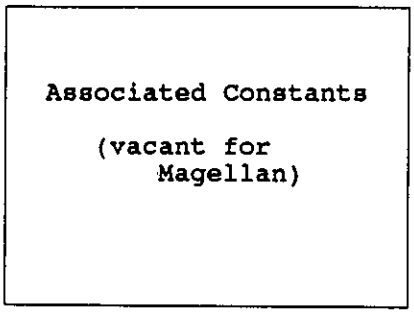


Figure 1-3 Associated Constants

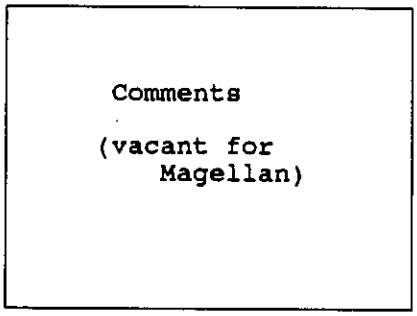


Figure 1-4 Comments

An actual sample of a Magellan SPK SFDU is shown in Figure 1-5. (The actual ephemeris data have been removed for brevity.) This SFDU was created with the MGNSPK program delivered for the SFOC Version 11 Build.

```
CCSD1Z00000100000835
NJPL1K00KL0000000674
DATA_OBJECT_TYPE=EPHEMERIS
PRODUCT_VERSION_TYPE=PREDICT
DATA_SET_NAME=MGN00999.TSP
MISSION_ID=4
MISSION_NAME=MAGELLAN
SPACECRAFT_NAME=MAGELLAN
SPACECRAFT_ID=18
PROCESS_TIME=1989-09-12T15:30:36.000
VERSION_ID=01
UPLOAD_ID=T0034A
ORBIT_NUMBER=00010
OBJECT=ORBIT_TIME_RANGE_GROUP
TIME_RANGE_NUMBER=1
ORBIT_START_TIME=1990-08-14T18:00:00.000
ORBIT_STOP_TIME=1990-08-14T21:10:00.000
END_OBJECT=ORBIT_TIME_RANGE_GROUP
OBJECT=MAPPING_TIME_RANGE_GROUP
TIME_RANGE_NUMBER=1
MAPPING_START_TIME=1990-08-14T18:10:00.000
MAPPING_STOP_TIME=1990-08-14T21:00:00.000
END_OBJECT=MAPPING_TIME_RANGE_GROUP
EFFECTIVE_TIME=1989-09-12T15:30:36.000
NAV_UNIQUE_ID="TEST                                    "
NJPL1S00KL0000000000
      ( vacant ASSOCIATED CONSTANTS section )
CCSD1S00000200000000
      ( vacant COMMENTS section )
CCSD1R00000300000082
DELIMITER=SMARKER
PRODUCT_NAME=NAIF_SPK_FILE
TYPE=NJPL1I000152
PROTOCOL=NONE
          .
          ( text format SPK file goes here )
          .
          .
CCSD1R00000300000048
DELIMITER=EMARKER
PRODUCT_NAME=NAIF_SPK_FILE
```

Figure 1-5 Sample of Magellan SPK SFDU

1.4.3.3 SPK File Contents Summary

The NAIF SPK file structure is intended to be sufficiently flexible to meet the science requirements of JPL flight projects and the Planetary Data System. Thus, SPK files may contain data in a variety of representations that yield states (Cartesian positions and velocities) for solar system objects, including spacecraft, planets, satellites, planet barycenters, comets, asteroids, Earth, and the Sun.

For the Magellan Project, SPK files contain:

- Chebyshev polynomials for the Sun, Earth-Moon barycenter, Earth, Moon, and Venus

- modified difference arrays for the Magellan spacecraft

These polynomial data can be used to compute the state of any of these objects with respect to any other object, in any reference coordinate frame supported by SPICELIB, NAIF's portable Fortran-77 toolkit library. The IAU J2000 reference frame is expected to be the normal choice for Magellan operations; B1950, FK4, and several JPL DE-nnn reference frames are alternate choices. See Appendix A for more information about reference frames.

1.4.3.4 Assumptions and Constraints

In general, SPK files can be created by anyone who has access to SPICELIB and to Appendices A and B. Some of the Magellan science teams may wish to create SPK files.

The SPK files made by the MGNSPK program must be created from ephemeris data produced per the spacecraft and planetary ephemeris formats specified in Applicable Document #4b. Only those SPK files created by MGNSPK are covered by this SIS.

It is assumed that the UNISYS computer to be used by the NAV Team will be equipped with the hardware and software needed to effect reliable FTP data transmissions over the SFOC TCP/IP channel of the ILAN to the NAV workstation. The same is assumed for the UNISYS computer on which the Key Times Files are produced.

Source NAVIO-format spacecraft ephemeris files must have sufficient overlap to ensure that any required SPK file can be made from a single spacecraft ephemeris: MGNSPK does not permit data from multiple spacecraft ephemeris files to be combined to make a single SPK file. Note that SPK files are required to include data covering one full orbit plus a ten percent overlap at both ends.

A valid Key Times File containing data for an orbit of interest must be available before an SPK file can be made for that orbit: the MGNSPK program has no provision for obtaining orbit and mapping start and stop times from an alternate source (such as user input).

Magellan NAV Team personnel must have some means to determine which Key Times File should be used to produce a given SPK file.

SECTION 2

ENVIRONMENT

2.1 HARDWARE CHARACTERISTICS AND LIMITATIONS

The system shall operate on a SFOC workstation, which shall include:

1. A full ANSI standard Fortran-77 compiler
2. A 400 MB disk capacity
3. A TCP/IP network interface, including FTP functionality
4. A printer
5. A 6250-bpi, 75-ips, industry standard half-inch tape drive

The tape drive will be used:

1. as the means for preparing SPK ephemeris file deliverables for the Magellan Data Management and Archive Team (DMAT).
2. to archive source NAVIO format P-files.
3. as a backup interface to the UNISYS for source P-files in the event of a failure of the TCP/IP network interface.

2.2 INTERFACE MEDIUM AND CHARACTERISTICS

Text format SPK SFDUs generated by this system will be transported to, and catalogued in, the SFOC Central Database (CDB) system using the CDB FTS service.

Text format SPK files required to be delivered to the DMAT will be written on 6250 bpi half-inch magnetic tape using the UNIX tar command.

2.3 INPUT/OUTPUT PROTOCOLS

2.3.1 Device Addressing

Not Applicable

2.3.2 Operating System Protocols

Operating system protocols for transferring SPK files to the CDB are handled by FTP. Files must be transferred using the FTP text mode. See Applicable Document #2 for information on FTP.

2.4 FAILURE PROTECTION, DETECTION, AND RECOVERY FEATURES

2.4.1 Backup Requirements

A tar tape is available as a backup transport capability.

2.4.2 Security/Integrity Measures

Normal SFOC/Magellan security provisions are applicable.

2.5 END-OF-FILE CONVENTION

The normal end of file convention for text files defined by the operating system of the NAV workstation will be used.

SECTION 3

DATA FLOW CHARACTERISTICS

3.1 OPERATIONAL CHARACTERISTICS

3.1.1 Generation Method and Frequency

The Magellan Project NAV Team will nominally generate one NAVIO format spacecraft ephemeris file each working day (five-day weeks). The exact details of the time span covered by each file -- the start and stop epochs relative to real time -- are complex and beyond the scope of this document. However, each file will cover a day or more into the past, and several days into the future (with additional coverage, as needed, to account for weekends).

Nominally, each NAVIO spacecraft ephemeris file will be used to create at least eight SPK files, each containing ephemeris data for a single orbit plus the required overlap. If the orbit covered by an SPK file is one for which tracking data were available, the SPK file is labeled as an "actual" file; otherwise, it is labeled as a "predict" file.

Both "actual" and "predict" files are produced by the staff of the Magellan Navigation Team using the program MGNSPK.

At least one "actual" SPK file must be generated for each orbit of the spacecraft. Actual files are included on the EDR tapes produced by the Magellan High Rate system. Predict files are used to make "temporary EDR" (TEDR) tapes.

3.1.2 Time Span of Product

Each actual or predict SPK file will cover one full orbit of the Magellan spacecraft, plus a user specified percentage overlap into the previous and subsequent orbits. The nominal amount of overlap for each end of the file is expected to be ten percent. The starting and ending epochs used in determining the coverage of a particular SPK file are obtained from the relevant Key Times File for that orbit.

3.1.3 Data Volume

An SPK file in text format and covering one full orbit plus ten percent at both ends will require approximately 155 kilobytes of disc storage. A binary version of an SPK file will be smaller than the text version by approximately a factor of three.

3.2 FLOW RATE

Nominally there will be eight transfers of "actual" SPK files to CDB each day, five days per week. Additional transfers will be needed to cover the sixteen orbits-worth of data generated during each weekend. In addition, some number of "predict" one-orbit SPK files will be transferred to the CDB.

Magellan Interface No. NAV-21 calls for daily delivery via magnetic tape (two copies) of the archival SPK files to the Magellan DMAT. The tapes will be produced on the NAV workstation and will be accompanied by a printout of the SFDU Catalog and History label section for each SPK file contained on the tape.

It has been proposed that the requirement for daily deliveries to DMAT be changed to a once-per-week delivery, thus reducing manpower and tape cost/storage requirements. The Magellan DMAT supports this proposal.

SECTION 4

DETAILED DATA OBJECT DEFINITION

4.1 Data Format and Definition

The NAIF SPK format is intended to serve many planetary science purposes -- it is not limited to the Magellan Flight Project. Consequently, documentation of the SPK file format and the underlying Double Precision Array File architecture are contained in a separate pair of NAIF documents included here as Appendices A and B, which are attached to this SIS for easy reference. (While the use of external references may present a small inconvenience to Magellan personnel, it allows the same documents to be used for all of NAIF's flight project and PDS applications.)

Appendices A and B also describe the NAIF portable Fortran-77 subroutines provided to read and write SPK files. The subroutines are a subset of SPICELIB, a multi-mission product which is distributed by SFOC to Magellan Project users of SPK files. This software is intended to provide a complete interface to SPK files, except for the two SFDU-related functions noted below.

4.2 SPK Utilities: Getting Ready To Use an SPK File

Magellan SPK files are delivered on EDR and TEDR tapes produced by the Magellan High Rate system. Before the files can be used to generate state vectors, they must be removed from the tape, converted to normal text format (minus SFDU scaffolding), and converted to binary format. NAIF provides a utility program, SPACIT, to perform the final step.

4.2.1 Separating a Text-Format SPK File From Other Products

Within an EDR or TEDR tape, the SPK SFDU and other data products are combined into a single SFDU. Because the SFDU architecture employed for Magellan is byte oriented, SPICELIB software is unable to find and access the SPK SFDU amongst the other SFDUs on the tape: the user must provide software to effect this separation.

4.2.2 Separating the SPK File From Its Own SFDU Envelope and Labels

The absence of a text-format option in the current SFDU methodology prohibits SPACIT from skipping through, or first removing, the SFDU and other labels that are attached at the front and the rear of the text-format SPK file. The user must provide software to remove these elements from the SPK files. The user may also have to restore the normal end-of-line convention appropriate for his host machine.

4.2.3 Converting a Text Format SPK FIle to Local Binary Format

The resulting text SPK file is converted to an equivalent binary file by the utility program SPACIT, which is described in Appendix C.

4.3 Differences Between NAVIO and SPK Files

Magellan SPK files contain a subset of the data found in the source NAVIO-format spacecraft and planet ephemeris files produced by the Magellan NAV Team. Specifically, SPK files contain only the data necessary to compute the states (Cartesian positions and velocities) of ephemeris objects as continuous functions of time. No additional information other than NAV Solution ID is retained from the source files, including the values of the constants used in the generation of the files.

For most of the planetary science community, these constants are not immediately useful. More important, the SPK format is designed to handle many types of data -- conic and equinoctial elements, discrete state vectors, and so on -- not just data from NAVIO spacecraft and ephemeris files. The inclusion of constants records in the SPK format would create more confusion than it would alleviate.

This is not to say that the states returned from SPK files are not equivalent to the states returned from the NAVIO source files -- they are identical within the bounds specified in Section 1.4.

4.4 Traceability

As described in Appendix A, every segment in every SPK file contains a 40-character segment identifier. The purpose of this identifier is to provide a link to the person or institution that originally produced the data contained in the segment.

In every Magellan SPK file produced by MGNSPK, this segment identifier string contains a NAV Solution ID label that uniquely identifies a particular source NAVIO-format spacecraft or planet ephemeris file; these files, and the data used to create them, should be archived by the Magellan NAV Team.

Appendix A

S- and P-Kernel (SPK) Specification and User's Guide

S- and P- Kernel (SPK)
Specification and User's Guide

Navigation Ancillary Information Facility

Prepared by
I.M. Underwood

NAIF Document No. 168.0
8 November 1989

Table of contents

Purpose

The purpose of this document is to describe SPK, the common file format for NAIF S- and P-kernel files, largely in support of other documents. The various mission-specific Software Interface Specification (SIS) documents for the SPK format refer to this document for details common to all all missions.

Intended audience

This document is intended for all users of S- and P-kernel files.

References

All references are to NAIF documents. The notation [Dn.m] refers to NAIF document number n, revision number m.

Introduction

In order to analyze the data returned by a spacecraft, such as Voyager, it is necessary to know, at any given epoch, the positions of the spacecraft and all the target bodies of interest. The purpose of the SPK—which stands for S(pacecraft) and P(lanet) Kernel—format is to allow ephemerides for any collection of solar system bodies to be combined under a common file format, and accessed by a common set of subroutines.

The S and P kernels are two of the five kernels at the heart of the SPICE concept being implemented by the Navigation Ancillary Information Facility (NAIF). Historically, ephemerides for spacecraft have been considered separately from those for planets and satellites. In fact, they are often generated through distinct processes, using different representations. However, there is no essential reason for keeping them separate: a spacecraft—like any planet, satellite, comet, or asteroid—has a center of mass, which at any given epoch has a position and velocity relative to the solar system barycenter. So the two kernels have been combined under a single format. For all practical purposes, they represent a single kernel.

The SPK format is supported by a collection of subroutines, called 'readers', which are part of the SPICELIB toolkit library. These subroutines support the following functions:

1. Make the ephemeris data in one or more SPK files available to a program.

2. Return the apparent, true, or geometric state (position and velocity) of one solar system body as seen from another.

The SPK format is one of the formats included under the DAF (Double precision Array File) architecture. Readers who are not familiar with that architecture are referred to [167.0], which describes the common aspects of all DAF formats, as well as a collection of SPICELIB subroutines that support the DAF architecture.

All of the DAF routines in SPICELIB may be used to create, populate, and manipulate SPK files. The values of ND and NI for the SPK format are ND=2 and NI=6.

Segments

An SPK file contains several 'segments'. Each segment contains ephemeris data sufficient to compute the geometric state (position and velocity) of one solar system body (the 'target') with respect to another (the 'center'), at any epoch throughout some finite interval of time.

Either body may be a spacecraft, a planet or planet barycenter, a satellite, a comet, an asteroid, or an aribitrary point for which an ephemeris has been calculated. Each body in the solar system is associated with a unique integer code. A list of names and codes for the planets, the major satellites, and several spacecraft can be found at the end of this document.

The states computed from the ephemeris data in a segment must be referenced to a single, recognized inertial reference frame. A list of recognized frames can be found at the end of this document.

The data in each segment are stored as a single array. The summary for the array, called a 'descriptor', has two double precision components:

1. The initial epoch of the interval for which ephemeris data are contained in the segment, in ephemeris seconds past Julian year 2000.
2. The final epoch of the interval for which ephemeris data are contained in the segment, in ephemeris seconds past Julian year 2000.

The descriptor has six integer components:

1. The NAIF integer code of the target.
2. The NAIF integer code of the center.
3. The NAIF integer code for the inertial reference frame.
4. The integer code for the representation (type of ephemeris data).
5. The initial address of the array.

6. The final address of the array.

The name of each array can contain up to 40 characters. This space must be used to store a 'pedigree' for the data in the array. The pedigree of a segment should allow a user to determine the conditions under which the data in the segment were generated. For example, given the pedigree of a segment containing an ephemeris for the Magellan spacecraft relative to Venus, a user should be able, by contacting the appropriate party (named in the pedigree), to determine the values of the various harmonic coefficients used to integrate that particular ephemeris.

Data types

The fourth integer component of the descriptor—the code for the representation, or 'data type'—is the key to the SPK format.

For purposes of determining the segment best suited to fulfill a particular request, all segments are treated equally. It is only when the data in a segment are to be evaluated—when a state vector is to be computed—that the type of data used to represent the ephemeris becomes important.

Because this step is isolated within a single low-level reader, SPKPV, new data types can be added to the SPK format without affecting application programs that use the higher level readers. SPKPV is designed so that the changes required to implement a new data type are minimal.

There are no real limits on the possible representations that can be used for ephemeris data. Users with access to data suitable for creating an ephemeris may to invent their own representations, adapting SPKPV accordingly.

However, only a small number of representations will be offcicially supported by NAIF. Only three representations are currently supported.

1. Modified difference arrays. Created by the JPL Orbit Determination Program (ODP), these are used primarily for spacecraft ephemerides.
2. Chebyshev polynomials for position. (Velocities are obtained by differentiation). These are used primarily for planet barycenters, and for satellites whose orbits are integrated.
3. Chebyshev polynomials for position and velocity. These are used primarily for satellites whose orbits are computed directly from analytic theories.

Subroutines

SPICELIB contains a family of subroutines that are designed specifically for use with SPK files. The name of each routine begins with the letters 'SPK', followed by a two- or three-character mnemonic. For example, the routine that returns the state of one body with respect to another is named SPKEZ, pronounced 'S-P-K-E-Z'. A complete list of mnemonics, translations, and calling sequences can be found at the end of this document.

Each subroutine is prefaced by a complete SPICELIB module header, which describes inputs, outputs, restrictions, and exceptions, discusses the context in which the subroutine should be used, and shows typical examples of its use. Any discussion of the subroutines in this document is intended as an introduction: the final documentation for any subroutine is its module header.

Whenever an SPK subroutine appears in an example, the translation of the mnemonic part of its name will appear to the right of the reference, in braces. For example,

```
CALL SPKLEF ( FNAME, HANDLE )                    { Load ephemeris file }
```

All subroutines and functions, including those whose names do not begin with 'SPK', are from SPICELIB.

Code examples will make use of the structured DO ... END DO and DO WHILE ... END DO statements supported by the VAX/VMS Fortran compiler. These statements are easily converted to the standard equivalents

```
        DO label var = e1, e2, e3
           stmt
label   CONTINUE
```

and

```
label   IF       ( expr )
       .THEN
           stmt
           GO TO label
        END IF
```

SPK readers are available to perform the following functions.

1. Determine the apparent, true, or geometric state of any body with respect to any other body.
2. Determine the apparent, true, or geometric state of any body with respect to an observer with an arbitrary (user-supplied) state.

3. Determine the geometric state of any body with respect to the solar system barycenter.
4. Determine the geometric state of a target body with respect to the center of motion for a particular segment.
5. Determine, from a list of SPK files supplied by the calling program, the files and segments best suited to fulfill a request for the state of a particular body.

These subroutines form a hierarchy: (1) uses (2) and (3); (2) uses (3); and (3) uses (4) and (5). Thus, routines (1), (2), and (3) can serve as starting points for more specialized routines that users may wish to write for themselves.

Computing states

SPKEZ is the most powerful of the SPK readers. It determines the apparent, true, or geometric state of one body (the target) as seen by a second body (the observer), provided that ephemeris data are available for both bodies.

```
CALL SPKEZ ( TARG, ET, FRAME, ABERR, OBS, STATE, LT )      { Easy state }
```

The subroutine accepts five inputs (target body, epoch, reference frame, aberration, and observing body) and returns two outputs (state of the target body as seen from the observing body, and one-way light-time from the target body to the observing body).

The target and observing bodies are identified by integer codes rather than names. For example, to determine the state of Triton (body 801) as seen from the Voyager-2 spacecraft (body -32),

```
CALL SPKEZ  ( 801, ET, FRAME, ABERR, -32, STATE, LT )      { Easy state }
```

By definition, the ephemerides in SPK files are continuous: the user can obtain states at any epoch within the interval of coverage. Epochs are always specified in ephemeris seconds past Julian year 2000. For example, to determine the state of Triton as seen from Voyager-2 at Julian Ephemeris Date 2447751.8293,

```
ET = ( 2447751.8293D0 - J2000() ) * SPD()

CALL SPKEZ  ( 801, ET, FRAME, ABERR, -32, STATE, LT )      { Easy state }
```

where the function J2000 returns Julian year 2000 (Julian Ephemeris Date 2451545.0) and the function SPD returns the number of seconds per Julian day (86400.0).

The ephemeris data in a single SPK file may be referenced to a number of different inertial reference frames. Because any two inertial frames differ by a simple rotation, states returned by SPKEZ do not have to be referenced to their native frames. The user can specify that states are to be returned in any of the recognized frames listed at the end of this document. For example, to determine the state of Triton as seen from Voyager-2, referenced to the IAU standard J2000 frame,

```
CALL SPKEZ  ( 801, ET, 'J2000', ABERR, -32, STATE, LT )     { Easy state }
```

SPKEZ returns apparent, true, or geometric states depending on the value of the aberration flag, ABERR.

Apparent states are corrected for planetary aberration, which is the composite of the apparent angular displacement produced by motion of the observer (stellar aberration) and the actual motion of the target body (correction for light-time). True states are corrected for light-time only. Geometric states are uncorrected.

Instead of using the potentially confusing terms terms 'true' and 'geometric' to specify the type of state to be returned, SPKEZ requires the specific corrections to be named. To compute apparent states, specify correction for both light-time and stellar aberration: 'LT+S'. To compute true states, specify correction for light-time only: 'LT'. To compute geometric states, specify no correction: 'NONE'.

In all cases, the one-way light-time from the target to the observer is returned along with the state. This is useful when determining the apparent orientation of the target body. For example, the following code fragment computes the sub-spacecraft latitude and longitude on Triton:

```
CALL SPKEZ  ( 801, ET, 'J2000', 'LT', -32, STATE, LT )     { Easy state }

CALL BODMAT ( 801, ET - LT, TIPM  )

CALL VMINUS (       STATE, STATE                    )
CALL MXV    ( TIPM, STATE, STATE                    )
CALL RECLAT (              STATE, RADIUS, LONG, LAT )
```

Subroutine BODMAT uses coefficients from the NAIF Planetary Constants kernel to compute the transformation from J2000 coordinates to pole and prime meridian coordinates for a specified body; VMINUS changes the direction of an arbitrary 3-vector; MXV multiplies a 3x3 matrix and a 3-vector; and RECLAT converts an arbitrary 3-vector into latitudinal coordinates.

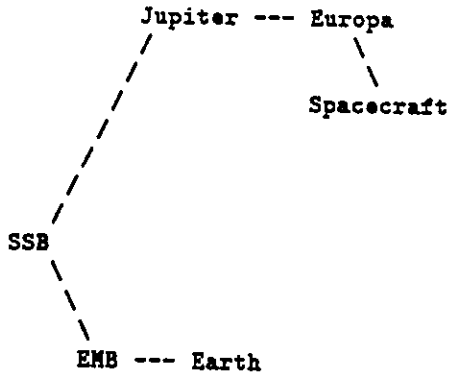
Loading files

Note that SPKEZ does not require the name of an SPK file as input. SPKEZ relies on a second routine, SPKLEF, to maintain a database of ephemeris files. The calling program indicates which files are to be used by passing their names to SPKLEF.

```
DO I = 1, N
   CALL SPKLEF ( EPHEM(I), HANDLE(I) )          { Load ephemeris file }
END DO
```

SPKLEF returns a DAF file handle for each file, which may be used to access the file directly using DAF subroutines. Once an SPK file has been loaded, it may be accessed by SPKEZ.

In general, a state state returned by SPKEZ is built from several more primitive states. Consider the following diagram, which shows some of the states that might be needed to determine the state of the Galileo spacecraft as seen from Earth:



(SSB and EMB are the solar system and Earth-Moon barycenters.)

Each state is computed from a distinct segment. The segments may reside in a single SPK file, or may be contained in as many as five separate files. For example, the segments needed to compute the Earth-spacecraft state shown above might come from the following set of files:

```
CALL SPKLEF ( 'barycenters.spk',    H(1) )      { Load ephemeris file }
CALL SPKLEF ( 'planet-centers.spk', H(2) )      { Load ephemeris file }
CALL SPKLEF ( 'satellites.spk',     H(3) )      { Load ephemeris file }
CALL SPKLEF ( 'spacecraft.spk',     H(4) )      { Load ephemeris file }
```

Or from the following set:

```
CALL SPKLEF ( 'earth.spk',       H(1) )      { Load ephemeris file }
CALL SPKLEF ( 'jupiter.spk',     H(2) )      { Load ephemeris file }
CALL SPKLEF ( 'spacecraft.spk', H(3) )       { Load ephemeris file }
```

An SPK file may contain any number of segments. In fact, the same file may contain overlapping segments: segments containing data for the same body over a common interval. When this happens, the latest segment in a file supersedes any competing segments earlier in the file. Similarly, the latest file loaded supersedes any earlier files. In effect, several loaded files become equivalent to one large file.

Unloading files

Sophisticated programs may need to open many files in the course of a single execution. A general-purpose 'solar system calculator', for example, may have to provide information about any body in the entire solar system; its usefulness is enhanced if users do not have to specify which files should be used.

On the other hand, the number of SPK files that may be open at any one time is limited. Such a program may need to unload some SPK files to make room for others. An SPK file may be unloaded by supplying its handle to subroutine SPKUEF. The sequence of statements shown below,

```
CALL SPKLEF ( 'file.a', HA )          { Load ephemeris file }
CALL SPKLEF ( 'file.b', HB )          { Load ephemeris file }
CALL SPKLEF ( 'file.c', HC )          { Load ephemeris file }
CALL SPKUEF (           HB )        { Unload ephemeris file }
CALL SPKLEF ( 'file.d', HD )          { Load ephemeris file }
CALL SPKUEF (           HC )        { Unload ephemeris file }
```

is equivalent to the following (shorter) sequence:

```
CALL SPKLEF ( 'file.a', HA )          { Load ephemeris file }
CALL SPKLEF ( 'file.d', HD )          { Load ephemeris file }
```

Lower-level readers

SPKEZ should be sufficient to handle the needs of most users. However, it is possible to exercise more direct control over the way states are computed. SPKEZ computes states using two readers of slightly less power. The first, SPKSSB, returns the state of a body with respect to the solar system barycenter. SPKEZ uses it to compute the state of the observer.

The second, SPKAPP, returns the state of a target body as seen from an observer. Where SPKEZ requires the integer code for an observer, SPKAPP requires the actual state of the observer with respect to the solar system barycenter. A single call to SPKEZ,

```
CALL SPKEZ ( 801, ET, 'J2000', 'LT+S', -32, STARG, LT )     { Easy state }
```

is equivalent to a pair of calls to SPKSSB and SPKAPP:

```
CALL SPKSSB ( -32,
              ET,
             'J2000',
              STOBS    )                  { Solar system barycenter }

CALL SPKAPP (  801,
               ET,
              'J2000',
               STOBS,
              'LT+S',
               STARG,
               LT       )                              { Apparent state }
```

One possible advantage of using SPKAPP directly is the ability to place an observer somewhere other than at the center of a body (for example, at a specified location on the surface of the Earth).

Primitive states

At the lowest level, it is possible to compute states without combining them at all. Given the handle and descriptor for a particular segment, subroutine SPKPV returns a state from that segment directly.

```
CALL SPKPV ( HANDLE,
             DESCR,
             ET,
             REF,
             STATE,
             CENTER   )                          { Position, velocity }
```

SPKPV is the most basic SPK reader. It rotates states into a specified reference frame, but does not combine them: it returns a state relative to the center whose integer code is stored in the descriptor for the segment. The user is responsible for using that state in the proper context.

The user is also responsible for using the DAF subroutines to determine the particular file and segment from which each state is to be computed.

If files have been loaded by previous calls to SPKLEF, it is possible to use the same choices that would normally be made by SPKEZ, SPKSSB, and SPKAPP. Subroutine SPKSFS selects, from the database of loaded files, the file handle and segment descriptor for the segment best suited to the request. If two segments from different files are suitable, SPKSFS selects the one from the file that was loaded later. If two segments from the same file are suitable, SPKSFS selects the one that is stored later in the file. The call

```
CALL SPKSFS ( 801, ET, HANDLE, DESCR, PDGREE, FOUND )   { Select file and
segment }
```

returns the handle, descriptor, and pedigree for the latest segment containing data for Triton at the specified epoch. SPKSFS maintains a buffer of segment descriptors and pedigrees, so it doesn't waste time searching the database for bodies it already knows about.

Example 1

The next several sections present sample programs to show how the SPK readers can be used to compute state vectors, and how those vectors can be used to compute derived quantities.

All subroutines and functions used in the examples are from SPICELIB. The convention of expanding SPK subroutine names will be dropped for these examples.

The first program is a program to compute the planetocentric latitude and longitude of the sub-observer point on a target body for any combination of observer, target, and epoch. (Note that planetocentric coordinates differ from planetographic and cartographic coordinates in that they are always right-handed, regardless of the rotation of the body.)

```
      PROGRAM LATLON

C
C     SPICELIB functions
C
      DOUBLE PRECISION  DPR

C
C     Variables
C
      CHARACTER*(32)    UTC

      DOUBLE PRECISION  ET
      DOUBLE PRECISION  LAT
      DOUBLE PRECISION  LON
      DOUBLE PRECISION  LT
      DOUBLE PRECISION  RADIUS
      DOUBLE PRECISION  STATE  (   6 )
      DOUBLE PRECISION  TIBF   ( 3,3 )

      INTEGER           OBS
      INTEGER           TARG

C
C     Load constants into the kernel pool. Two files are
C     needed. The first ('time.ker') contains the dates
C     of leap seconds and values for constants needed to
C     compute the difference between UTC and ET at any
C     epoch. The second ('trans.ker') contains IAU values
C     needed to compute transformations from inertial
C     (J2000) coordinates to body-fixed (pole and prime
C     meridian) coordinates for the major bodies of the
C     solar system. (These files, or their equivalents,
C     are normally distributed along with SPICELIB.)
C
      CALL CLPOOL
      CALL LDPOOL ( 'TIME.KER'  )
      CALL LDPOOL ( 'TRANS.KER' )
```

```
C
C     Several ephemeris files are needed. Most contain data for
C     a single planetary system ('jupiter.ker', 'saturn.ker',
C     and so on). Some contain data for spacecraft ('vgr1.ker',
C     'mgn.ker').
C
      CALL SPKLEF ( 'MERCURY.SPK' )
      CALL SPKLEF ( 'VENUS.SPK'   )
      CALL SPKLEF ( 'EARTH.SPK'   )
      CALL SPKLEF ( 'MARS.SPK'    )
      CALL SPKLEF ( 'JUPITER.SPK' )
      CALL SPKLEF ( 'SATURN.SPK'  )
      CALL SPKLEF ( 'URANUS.SPK'  )
      CALL SPKLEF ( 'NEPTUNE.SPK' )
      CALL SPKLEF ( 'PLUTO.SPK'   )
      CALL SPKLEF ( 'VGR1.SPK'    )
      CALL SPKLEF ( 'VGR2.SPK'    )
      CALL SPKLEF ( 'MGN.SPK'     )
      CALL SPKLEF ( 'GLL.SPK'     )

C
C     Inputs are entered interactively. The user enters three
C     items: the code for the observer (an integer), the code
C     for the target (an integer), and the epoch at which the
C     sub-observer point is to be computed (a free-format string).
C
C     The epoch must be converted to ephemeris time (ET).
C
      DO WHILE ( .TRUE. )

         WRITE (*,*)
         WRITE (*,*) 'Observer?'
         READ  (*,*)  OBS

         WRITE (*,*)
         WRITE (*,*) 'Target?'
         READ  (*,*)  TARG

         WRITE (*,*)
         WRITE (*,*)        'Epoch (UTC)?'
         READ  (*,FMT='(A)') UTC

         CALL UTC2ET ( UTC, ET )

C
C        Compute the true state (corrected for light-time)
C        of the target as seen from the observer at the
C        specified epoch. Do all computations in standard
C        (J2000) coordinates.
C
         CALL SPKEZ ( TARG, ET, 'J2000', 'LT', OBS, STATE, LT )

C
C        Compute the transformation from inertial to body-fixed
C        coordinates (TIBF) at the specified epoch minus the one-way
C        light-time from the target to the observer. (The target
C        continues to rotate as light reflected from it races
C        toward the observer.) BODMAT retrieves the necessary
C        coefficients from the kernel pool.
```

```
C
      CALL BODMAT ( TARG, ET - LT, TIBF  )

C
C     Rotate the state into the body-fixed frame. Although we
C     computed the vector FROM the observer TO the target in
C     order to get the proper light-time correction, we need
C     the vector FROM the target TO the observer in order to
C     compute latitude and longitude. So reverse it.
C
      CALL VMINUS (       STATE, STATE )
      CALL MXV    ( TIBF, STATE, STATE )

C
C     Convert from rectangular coordinates to latitiude and
C     longitude, then from radians to degrees for output.
C
      CALL RECLAT ( STATE, RADIUS, LON, LAT )

      WRITE (*,*)
      WRITE (*,*) 'Sub-observer latitude (deg): ', LAT * DPR()
      WRITE (*,*) '             longitude     : ', LON * DPR()
      WRITE (*,*)
      WRITE (*,*) 'Range to target (km)       : ', RADIUS
      WRITE (*,*) 'Light-time (sec)           : ', LT
      WRITE (*,*)

C
C     Get the next set of inputs.
C
   END DO

   END
```

Example 2

The second program computes the same quantities as the first. However, this program assumes that the observer is always the Magellan spacecraft and the target is always Venus. It also ignores light-time from the planet to the spacecraft. These restrictions allow a more primitive reader, SPKPV, to be substituted for the more general reader, SPKEZ.

SPKEZ would compute the state of the spacecraft with respect to the planet by computing the states of both bodies with respect to the solar system barycenter. SPKPV returns this same state, but does it directly—making the second program much faster than the first.

But the second program is much less flexible. For example, if the spacecraft ephemeris contains cruise data (describing the motion of the spacecraft relative to the solar system barycenter instead of the planet center), the program would produce incorrect results.

Furthermore, the program cannot easily be generalized to work for other orbiters. The motion of the Galileo spacecraft, for instance, would normally be known relative to the Jupiter barycenter, not to the planet itself.

```
      PROGRAM FASTER

C
C     SPICELIB functions
C
      DOUBLE PRECISION  DPR

C
C     Definitions
C
      INTEGER           MGN
      PARAMETER       ( MGN = -18 )

      INTEGER           VENUS
      PARAMETER       ( VENUS = 299 )

C
C     Variables
C
      CHARACTER*(40)    PDGREE
      CHARACTER*(32)    UTC

      DOUBLE PRECISION  DESCR  (   5 )
      DOUBLE PRECISION  ET
      DOUBLE PRECISION  LAT
      DOUBLE PRECISION  LON
      DOUBLE PRECISION  RADIUS
      DOUBLE PRECISION  STATE  (   6 )
      DOUBLE PRECISION  TIBF   ( 3,3 )

      INTEGER           CENTER
```

```
      INTEGER           HANDLE

      LOGICAL           FOUND

C
C     Load constants into the kernel pool. Two files are
C     needed. The first ('time.ker') contains the dates
C     of leap seconds and values for constants needed to
C     compute the difference between UTC and ET at any
C     epoch. The second ('venus.ker') contains IAU values
C     needed to compute the transformation from inertial
C     (J2000) coordinates to body-fixed (pole and prime
C     meridian) coordinates for Venus.
C
      CALL CLPOOL
      CALL LDPOOL ( 'TIME.KER'  )
      CALL LDPOOL ( 'VENUS.KER' )

C
C     Only one ephemeris file is needed. This contains data for
C     the Magellan spacecraft relative to Venus. The states of
C     other bodies are not needed.
C
      CALL SPKLEF ( 'MGN.SPK' )

C
C     Inputs are entered interactively. The user enters only the
C     epoch at which the sub-spacecraft point is to be computed
C     (a free-format string).
C
C     The epoch must be converted to ephemeris time (ET).
C
      DO WHILE ( .TRUE. )

         WRITE (*,*)
         WRITE (*,*)        'Epoch (UTC)?'
         READ  (*,FMT='(A)') UTC

         CALL UTC2ET ( UTC, ET )

C
C        Because the ephemeris file might contain many segments
C        for the spacecraft, we need to select the proper segment
C        each time a state is computed.
C
C        For now, we will assume that a segment is found. A more
C        careful program would check this each time. (If FOUND is
C        ever false, the data needed to respond to the user's
C        request are not available, and the program should take
C        appropriate action.)
C
         CALL SPKSFS ( MGN, ET, HANDLE, DESCR, PDGREE, FOUND )

C
C        Compute the geometric state (uncorrected for light-time)
C        of the spacecraft as seen from the planet. We can compute
C        this directly because light-time is being ignored.
C        Do all computations in standard (J2000) coordinates,
C
```

```
C        For now, we will assume that CENTER is always Venus
C        (2 or 299). A more careful program would check this
C        each time.
C
         CALL SPKPV ( HANDLE, DESCR, ET, 'J2000', STATE, CENTER )

C
C        Compute the transformation from inertial to body-fixed
C        coordinates (TIBF) at the specified epoch, ignoring
C        light-time.
C
         CALL BODMAT ( VENUS, ET, TIBF  )

C
C        Rotate the state into the body-fixed frame.
C
         CALL MXV ( TIBF, STATE, STATE )

C
C        Convert from rectangular coordinates to latitiude and
C        longitude, then from radians to degrees for output.
C
         CALL RECLAT ( STATE, RADIUS, LON, LAT )

         WRITE (*,*)
         WRITE (*,*) 'Sub-spacecraft latitude (deg): ', LAT * DPR()
         WRITE (*,*) '                longitude     : ', LON * DPR()
         WRITE (*,*)

C
C        Get the next input epoch.
C

      END DO

      END
```

Example 3

The third program determines epochs when one target body (spacecraft, planet, or satellite) is occulted by or in transit across another target body as seen from an arbitrary observer. It is similar in both form and generality to the first program.

```
      PROGRAM OCCTRN

C
C     SPICELIB functions
C
      DOUBLE PRECISION      SUMAD
      DOUBLE PRECISION      VNORM
      DOUBLE PRECISION      VSEP

C
C     Variables
C
      CHARACTER*(32)        UTC

      DOUBLE PRECISION      AVG
      DOUBLE PRECISION      D       ( 2   )
      DOUBLE PRECISION      ET
      DOUBLE PRECISION      R       ( 2   )
      DOUBLE PRECISION      RADII   ( 3   )
      DOUBLE PRECISION      S       ( 6,2 )
      DOUBLE PRECISION      SEP

      INTEGER               I
      INTEGER               OBS
      INTEGER               T       ( 2 )

C
C     Load constants into the kernel pool. Two files are
C     needed. The first ('time.ker') contains the dates
C     of leap seconds and values for constants needed to
C     compute the difference between UTC and ET at any
C     epoch. The second ('radii.ker') contains values
C     for the tri-axial ellipsoids used to model the major
C     major bodies of the solar system.
C
      CALL CLPOOL
      CALL LDPOOL ( 'TIME.KER'  )
      CALL LDPOOL ( 'RADII.KER' )

C
C     Several ephemeris files are needed. Most contain data for
C     a single planetary system ('jupiter.ker', 'saturn.ker',
C     and so on). Some contain data for spacecraft ('vgr1.ker',
C     'mgn.ker').
C
      CALL SPKLEF ( 'MERCURY.SPK' )
      .
      .
      CALL SPKLEF ( 'GLL.SPK'     )
```

```
C
C     Inputs are entered interactively. The user enters four
C     items: the code for the observer (an integer), the codes
C     for two target bodies (integers), and the epoch at which the
C     sub-observer point is to be computed (a free-format string).
C
C     The epoch must be converted to ephemeris time (ET).
C
      DO WHILE ( .TRUE. )

         WRITE (*,*)
         WRITE (*,*) 'Observer?'
         READ  (*,*)  OBS

         WRITE (*,*)
         WRITE (*,*) 'Targets?'
         READ  (*,*)  T(1), T(2)

         WRITE (*,*)
         WRITE (*,*)        'Epoch (UTC)?'
         READ  (*,FMT='(A)') UTC

         CALL UTC2ET ( UTC, ET )

C
C        Get the apparent states of the target objects as seen from
C        the observer. Also get the apparent radius of each object
C        from the kernel pool. (Use zero radius for any spacecraft;
C        use average radius for anything else.)
C
C           T(i)        is the ID code of the i'th target.
C           S(1-6,i)    is the apparent state of the i'th target.
C           D(i)        is the apparent distance to the i'th target.
C           R(i)        is the apparent radius of the i'th target.
C
C        Function VNORM returns the Euclidean norm (magnitude) of
C        a three-vector.
C
C        Function SUMAD returns the sum of the elements in a
C        double precision array.
C
         DO I = 1, 2
            CALL SPKEZ ( T(I), ET, 'J2000', 'LT+S', OBS, S(1,I), LT )
            D(I) = VNORM ( S(I) )

            IF ( T(I) .LT. 0 ) THEN
               R(I) = 0.D0

            ELSE
               CALL BODVAR ( T(I), 'RADII', DIM, RADII )
               AVG  = SUMAD ( RADII, 3 ) / 3.D0
               R(I) = ASIN ( AVG / D(I) )
            END IF
         END DO

C
C        Determine the separation between the two bodies. If the
C        separation between the centers is greater than the sum of
```

```
C        the apparent radii, then the target bodies are clear of
C        each other.
C
C        Function VSEP returns the angle of separation between
C        two three-vectors.
C
         SEP = VSEP ( S(1,1), S(1,2) ) - ( R(1) + R(2) )

         IF ( SEP .GT. 0 ) THEN

            WRITE (*,*)
            WRITE (*,*) 'Clear.'

C
C        Otherwise, the smaller body is either occulted or in transit.
C
         ELSE

            IF ( R(1) .LT. R(2) ) THEN
               IF ( D(1) .LT. D(2) ) THEN
                  WRITE (*,*)
                  WRITE (*,*) T(1), ' in transit across ', T(2)
               ELSE
                  WRITE (*,*)
                  WRITE (*,*) T(1), ' occulted by ', T(2)
               END IF

            ELSE
               IF ( D(1) .LT. D(2) ) THEN
                  WRITE (*,*)
                  WRITE (*,*) T(2), ' occulted by ', T(1)
               ELSE
                  WRITE (*,*)
                  WRITE (*,*) T(2), ' in transit across ', T(1)
               END IF
            END IF

         END IF

C
C        Get the next set of inputs.
C
      END DO

      END
```

Test program

A standard test program is provided to allow users to verify that the SPK readers have been installed correctly in any new environment. The program computes a series of state vectors, using data from a standard test ephemeris file, and compares the results with state vectors computed earlier in the NAIF development environment.

If the states agree to within some specified tolerance (which can be changed by the user), the program simply reports that the SPK readers are working correctly. Otherwise, it prints out diagnostic information that should be forwarded to NAIF.

Supported data types

The following representations, or data types, for are currently supported by the SPK routines in SPICELIB.

1. Modified Difference Arays.

   These are difference tables, normally used for spacecraft ephemerides.

2. Chebyshev polynomials (position only).

   These are sets of coefficients for the x, y, and z components of the body position. The velocity of the body is obtained by differentiation. This data type is normally used for planet barycenters, and for satellites whose orbits are integrated.

3. Chebyshev polynomials (position and velocity).

   These are sets of coefficients for the x, y, and z components of the body position, and for the corresponding components of the velocity. This data type is normally used for for satellites whose orbits are computed directly from theories.

Because SPK files are (DAF) array files, each segment is stored as an array. Each array corresponding to a particular data type has a particular internal structure. These structures are described below.

Type 1: Modified Difference Arrays

Each segment containing Modified Difference Arrays contains an arbitrary number of logical records, each containing difference line coefficients valid up to some final epoch, along with the state at that epoch. Subroutine SPKE01 contains the algorithm used to construct a state from a particular record and epoch. The contents of the records themselves are described in [163.0].

The records within a segment are ordered by increasing final epoch. A segment of this type is structured as follows:

```
Record 1
Record 2
 ...
Record N
Epoch 1
Epoch 2
 ...
Epoch N
Directory epoch 1
Directory epoch 2
 ...
Directory epoch N/100
N
```

Records 1 through N contain the difference line coefficients and other constants needed to integrate state data. Each one of these records contains 71 double precision numbers.

The list of final epochs for the records is stored immediately after the last record.

Following the list of epochs is a second list, the 'directory', containing every 100th epoch from the previous list. If there are N epochs, there will be N/100 directory epochs. If there are fewer than 100 epochs, then the segment will not contain any directory epochs. Directory epochs are used to speed up access to desired records.

The final element in the segment is the number of records contained in the segment.

The index of the record corresponding to a particular epoch is the index of the first epoch not less than the target epoch.

Type 2: Chebyshev (position only)

Each segment containing Chebyshev polynomials for position only contains an arbitrary number of logical records, each containing a set of Chebyshev coefficients valid throughout an interval of fixed length. Subroutine SPKE02 contains the algorithm used to construct a state from a particular record and epoch.

The records within a segment are ordered by increasing initial epoch. All records contain the same number of coefficients. A segment of this type is structured as follows:

```
Record 1
Record 2
 ...
Record N
INIT
INTLEN
RSIZE
N
```

A four-number 'directory' at the end of the segment contains the information needed to determine the location of the record corresponding to a particular epoch.

1. INIT is the initial epoch of the first record, in ephemeris seconds past J2000.
2. INTLEN is the length of the interval covered by each record, in seconds.
3. RSIZE is the total size of (number of array elements in) each record.
4. NREC is the number of records contained in the segment.

Each record is structured as follows:

```
MID
RADIUS
X coefficients
Y coefficients
Z coefficients
```

The first two elements in the record, MID and RADIUS, are the midpoint and radius of the time interval covered by coefficients in the record. These are used as parameters to perform transformations between the domain of the record and the domain of Chebyshev polynomials (which is always the same). Each polynomial in the segment is of the same degree, which is given by

```
(RSIZE - 2) / 3
```

Type 3: Chebyshev (position and velocity)

Segments containing Chebyshev polynomials for both position and velocity differ from segments of the previous type in the following ways:

First, coefficients are provided for position and velocity. (Subroutine SPKE03 contains the algorithm used to construct a state from a particular record and epoch.) Therefore, each record contains six sets of coefficients instead of three:

```
MID
RADIUS
X  coefficients
Y  coefficients
Z  coefficients
X' coefficients
Y' coefficients
Z' coefficients
```

Second, the degree of each polyniomial is given by

```
(RSIZE - 2) / 6
```

Body codes

In theory, a unique integer can be asssigned to each body in the solar system, including interplanetary spacecraft. The SPK routines in SPICELIB use integer codes instead of names to refer to ephemeris bodies for three reasons.

1. Space

   Integers code are smaller than alphanumeric names.

2. Uniqueness

   The names of some satellites conflict with the names of some asteroids and comets. Also, some satellites are commonly referred to by names other than those approved by the IAU.

3. Context

   The type of a body (barycenter, planet, satellite, comet, asteroid, or spacecraft) and the system to which it belongs (Earth, Mars, Jupiter, Saturn, Uranus, Neptune, or Pluto) can be recovered algorithmically from the integer code assigned to a body. This is not generally true for names.

The smallest positive codes are reserved for the Sun and the planet barycenters:

```
 0     Solar system barycenter
 1     Mercury           "
 2     Venus             "
 3     Earth             "
 4     Mars              "
 5     Jupiter           "
 6     Saturn            "
 7     Uranus            "
 8     Neptune           "
 9     Pluto             "
10     Sun               "
```

The code for a satellite is normally computed by adding its IAU designation to 100 times the code for its barycenter. For example, Ananke, the 12th satellite of Jupiter (JXII), is body number 512. This system works for all satellites discovered prior to 1989—numbers for the new satellites of Neptune have not yet been determined by the IAU.

```
301   Moon

401   Phobos
402   Deimos

501   Io
502   Europa
503   Ganymede
504   Callisto
505   Amalthea
506   Himalia
507   Elara
508   Pasiphae
509   Sinope
510   Lysithea
511   Carme
512   Ananke
513   Leda
514   Thebe       (1979J2)
515   Adrastea    (1979J1)
516   Metis       (1979J3)

601   Mimas
602   Enceladus
603   Tethys
604   Dione
605   Rhea
606   Titan
607   Hyperion
608   Iapetus
609   Phoebe
610   Janus       (1980S1)
611   Epimetheus  (1980S3)
612   Helene      (1980S6)
```

```
613   Telesto     (1980S13)
614   Calypso     (1980S25)
615   Atlas       (1980S28)
616   Prometheus  (1980S27)
617   Pandora     (1980S26)

701   Ariel
702   Umbriel
703   Titania
704   Oberon
705   Miranda
706   Puck        (1985U1)
707   Portia      (1986U1)
708   Rosalind    (1986U2)
709   Juliet      (1986U3)
710   Cressida    (1986U4)
711   Belinda     (1986U5)
712   Desdemona   (1986U6)
713   Cordelia    (1986U7)
714   Ophelia     (1986U8)
715   Bianca      (1986U9)

801   Triton
802   Nereid

901   Charon      (1978P1)
```

A planet is always considered to be the 99th satellite of its own barycenter. For example, Jupiter is body number 599. Mercury and Venus have no satellites, so bodies 199 and 299 are the same as bodies 1 and 2. For all practical purposes, this is true for Mars (499 and 4) as well.

```
199     Mercury
299     Venus
399     Earth
499     Mars
599     Jupiter
699     Saturn
799     Uranus
899     Neptune
999     Pluto
```

Negative codes are reserved for spacecraft. The code assigned to a particular spacecraft is normally just the negative of the code assigned to the same spacecraft by JPL's Deep Space Network (DSN). (Because the DSN recycles spacecraft numbers, this may not always be possible.) SPK codes have been assigned for the following spacecraft:

```
-12   Pioneer 12 (Venus orbiter)
-18   Magellan
-27   Viking 1 orbiter
-30   Viking 2 orbiter
-31   Voyager 1
```

```
-32   Voyager 2
-77   Galileo orbiter
-94   Mars observer
```

Codes for numbered asteroids from the JPL Asteroid and Comet Catalog [166.0] are created by multiplying the asteroid number by 10000 and adding 1010. For example, asteroid Yeomans (2956) is body number 29561010.

Numbers for periodic comets from the same catalog are created by mutliplying the comet number by 10000 and adding 2010. For example, the 1986 apparition of comet Halley (20181) is body number 201812010. Because the catalog may contain several apparitions for a single comet, and because catalog numbers are changed each time the catalog is updated, comet numbers generated by this system are not unique. So far, this has not caused any practical difficulties; however, if no standard numbering system is imposed on the catalog, NAIF will implement one independently.

Reference frames

Every state vector returned from an ephemeris must be referenced to a recognized inertial reference frame. This requirement ensures that primitive states from different ephemerides can be combined to create more general states. (By forbidding states referenced to dynamically defined frames, this requirement also ensures that the ephemeris data stored in a file cannot be modified by definitions stored outside the file.)

The inertial reference frames to which states may be referenced fall into three general catagories.

1. New

   New frames are identical or nearly identical to the frame defined by the Earth mean equator and dynamical equinox of Julian year 2000, adopted by the IAU as the standard reference frame for all publications. The frame defined by the (as yet unreleased) FK5 star catalog will be in this category.

2. Old

   Old frames are close to the frame defined by the Earth mean equator and dynamical equinox of Besselian year 1950. The frames in this category are often referred to collectively by the name 'EME50', although individual frames may be offset from each other by as much as half a second of arc. The name 'B1950' is reserved for the frame defined by applying the J2000 precession model [164.0] to the J2000 frame. The frame defined by the FK4 star catalog is offset from the B1950 frame

by about half a second of arc. Several of the frames in this category are defined by one of the Developmental Ephemeris (DE) files created by Standish at JPL. These may be offset from the FK4 frame by anywhere from a tenth to a hundredth of a second of arc [165.0].

3. Other

   Other frames may be defined as needed. Currently, only one frame does not fit into either of the previous categories: the Galactic (System II) reference frame.

The rotation between any two recognized frames can be obtained from subroutine IRFROT in SPICELIB by supplying the indexes of the frames taken from the following list:

```
Index   Name       Description
-----   --------   ----------------------------------------
    1   J2000      Earth mean equator, dynamical equinox of J2000
    2   B1950      Earth mean equator, dynamical equinox of B1950
    3   FK4        Fundamental Catalog (4)
    4   DE-118     JPL Developmental Ephemeris (118)
    5   DE-96      JPL Developmental Ephemeris ( 96)
    6   DE-102     JPL Developmental Ephemeris (102)
    7   DE-108     JPL Developmental Ephemeris (108)
    8   DE-111     JPL Developmental Ephemeris (111)
    9   DE-114     JPL Developmental Ephemeris (114)
   10   DE-122     JPL Developmental Ephemeris (122)
   11   DE-125     JPL Developmental Ephemeris (125)
   12   DE-130     JPL Developmental Ephemeris (130)
   13   GALACTIC   Galactic System II
   14   DE-200     JPL Developmental Ephemeris (200)
   15   DE-202     JPL Developmental Ephemeris (202)
```

(The module header for IRFROT always contains the definitive list of recognized frames.) For example, to rotate a position vector from FK4 coordinates to J2000 coordinates,

```
CALL IRFROT ( 3, 1, ROT           )
CALL MXV    (       ROT, OLD, NEW )
```

(ROT is a 3-by-3 matrix, OLD and NEW are 3-vectors; subroutine MXV multiplies a matrix and a vector to produce a vector.)

Two additional subroutines can be used to convert names to numbers (indexes) and vice versa. To find the index of the DE-125 frame,

```
CALL IRFNUM ( 'DE-125', NUMBER )
```

To find the name corresponding to index 6,

```
CALL IRFNAM ( 6, NAME )
```

Summary of mnemonics

SPICELIB contains a family of subroutines that are designed specifically for use with SPK files. The name of each routine begins with the letters 'SPK', followed by a two- or three-character mnemonic. For example, the routine that returns the state of one body with respect to another is named SPKEZ, pronounced 'S-P-K-E-Z'. The following is a complete list of mnemonics and translations, in alphabetical order.

```
APP     Apparent state
E01     Evaluate record, type 01
E02     Evaluate record, type 02
E03     Evaluate record, type 03
EZ      Easy state
LEF     Load ephemeris file
PV      Position, velocity
R01     Read record, type 01
R02     Read record, type 01
R03     Read record, type 01
SFS     Select file and segment
SSB     Solar system barycenter
UEF     Unload ephemeris file
```

Summary of calling sequences

The calling sequences for the SPK subroutines are summarized below. Subroutines are grouped by function.

Loading, unloading files:

```
SPKLEF ( FNAME, HANDLE )
SPKUEF (        HANDLE )
```

Computing states:

```
SPKEZ  ( TARGET, ET, REF,         ABERR, OBS, STATE, LT )
SPKAPP ( TARGET, ET, REF, STOBS, ABERR,       STATE, LT )
SPKSSB ( TARGET, ET, REF,                     STATE     )

SPKPV  ( HANDLE, DESCR, ET, REF, STATE, CENTER )
```

Selecting files, segments:

```
SPKSFS ( TARGET, ET, HANDLE, DESCR, PDGREE, FOUND )
```

Reading, evaluating records:

```
SPKR01 ( HANDLE, DESCR, ET, RECORD         )
SPKE01 (                ET, RECORD, STATE )

SPKR02 ( HANDLE, DESCR, ET, RECORD         )
SPKE02 (                ET, RECORD, STATE )

SPKR03 ( HANDLE, DESCR, ET, RECORD         )
SPKE03 (                ET, RECORD, STATE )
```

Appendix B

Double Precision Array Files (DAF) Specification and User's Guide

Double Precision Array Files (DAF)
Specification and User's Guide

Navigation Ancillary Information Facility

Prepared by
I.M. Underwood

NAIF Document No. 167.0
8 November 1989

Table of contents

Purpose

The purpose of this document is to describe DAF, an architecture for files that store double precision arrays, largely in support of other documents. For example, the NAIF S-, P-, and C-kernel file formats depend on the DAF architecture and associated software. The Software Interface Specification (SIS) documents for these formats refer to this document for details common to all three formats.

Intended audience

This document is intended for sophisticated users of DAF-based file formats, including the S-, P-, and C-kernel formats already mentioned. It is also intended for users of the SPICELIB toolkit library who wish to create their own DAF-based file formats.

Casual users of DAF-based formats will not generally need to understand the material presented in this document. For example, users of NAIF S-, P-, and C-kernel files who wish to derive state vectors and pointing angles from those files will normally do so using only subroutines and programs designed specifically for those formats.

Introduction

DAF—which stands for 'Double precision Array File'—is a file architecture that provides the advantages of arrays and direct access files without incurring the disadvantages of either one.

The architecture is supported by a set of Fortran-77 subroutines, part of the NAIF SPICELIB toolkit library.

DAF is called an architecture instead of a format because it includes an entire family of file formats, each of which is characterized by a pair of parameters. A file that conforms to any one of these formats is called an 'array file'.

As its name implies, an array file contains arrays. A single array file can contain any number of double precision arrays. Each of these arrays can contain an arbitrary number of elements. (Array elements must be 'pure' double precision numbers. That is, they may not contain equivalenced or encoded integer or character values.)

The DAF subroutines in SPICELIB support the following operations:

1. Create, open, or close an array file.

2. Add a new array to a file.

3. Locate an array within a file, either by index or by using descriptive information about the array.

4. Access—that is, retrieve or update—any contiguous set of elements in an array.

5. Convert a binary (direct access) array file to an equivalent text file, suitable for transfer over a network environments based on other CPUs, operating systems, or compilers.

6. Convert a text array file back into an equivalent binary file.

The last two functions make array files, like the SPICELIB subroutines, portable to any environment that supports the ANSI standard Fortran-77.

DAF-based formats

The parameters—ND and NI—that define a particular format within the DAF architecture do so by defining the amount and kinds of descriptive information that can be associated with each of the arrays in a particular file. ND and NI stand for 'Number of Double precision components' and 'Number of Integer components' respectively.

Values for ND and NI are fixed at the time an array file is created. Any two array files that have the same values for ND and NI can be thought of as having the same 'format'. (This does not guarantee that the arrays in the files contain the same kinds of information, only that they could be stored in the same file.)

Each array stored in an array file is 'described', in part, by ND double precision numbers and NI integer numbers, which are stored separately from the array. Most of the details of this 'description'—how many numbers are needed, and what they contain—are left to the designer of the format.

The double precision numbers could include limits (the smallest and largest values in the array), a range of epochs throughout which the elements may be used, or statistics (the mean, median, and standard deviation of the elements).

The integer numbers could include contextual information (case number, identification codes for related objects or arrays) and conditional information (flags to indicate whether the array is unsorted, sorted by increasing or decreasing magnitude, or marked for deletion). Some integer numbers are used to keep track of the location of the array within the file.

Each array in an array file is is also described by NC characters of alphanumeric information, where NC is a function of ND and NI. (NC is the smallest

multiple of eight not smaller than 8(ND) + 4(NI).) Alphanumeric information may include names, archive codes, historical information, or anything else that is not easily encoded as double precision or integer numbers.

The double precision and integer numbers that describe each array are 'packed', or equivalenced, into an auxiliary double precision array before they are stored in the file. The auxiliary array is called the 'summary' of the associated array. The individual (unpacked) numbers are called the 'components' of the summary.

(The first ND elements of the summary contain the double precision components of the summary. Each of the remaining elements contains a pair of integer components. If NI is odd, the final element of the summary contains a single integer component.)

The alphanumeric characters that describe each array are stored in a single character string, called the 'name' of the array.

Addresses

The location of each array in an array file is defined by a pair of numbers, called the 'initial address' and 'final address' of the array.

The term 'address' refers to a particular way of looking at an array file. Every array file is actually a standard Fortran-77 direct access file, with a particular record length. (Every array file has the same record length.) Each record is capable of storing up to 128 double precision numbers.

It is convenient, however, to think of an array file as a numbered collection of slots, 'words', each large enough to hold one double precision number. Words 1 through 128 are located in the first record of the file; words 129 through 256 are located in the second record; and so on. The number of each word is called the 'address' of the word within the file.

Any pair of addresses defines a contiguous set of words, which may fall within a single physical record or span several thousand of them. The elements of each array in an array file are stored in just such a set. The address of the first element is the 'initial address' of the array. The address of the final element is the 'final address' of the array.

The initial and final addresses of an array are always the final two integer components of the summary for the array.

Lists

It is a simplification, but a useful one, to say that the arrays in an array file form a doubly-linked list. Each new array added to a file is placed at the tail of this list.

Because the list is doubly-linked, the head and tail of the list can be located immediately. The arrays can be located by moving a pointer through the list, in either direction, one array at a time.

At any time, the summary and name of the array at which the pointer is currently pointing can be retrieved and examined to determine whether the array is of interest. If it is, the initial and final addresses (the final two integer components of the summary) may be used to access—retrieve or update—the entire array, or any contiguous set of elements therein.

For example, if BEGIN and END are the initial and final addresses of an array, the first ten elements of the array can be retrieved by asking for the elements stored in addresses BEGIN through BEGIN+9. The middle element of the array can be retrieved by asking for the element stored in address (BEGIN+END)/2.

Read and write access

Array files may be opened for two kinds of access: read and write. A file opened for read access cannot be changed, either by adding a new array or by updating an existing one. Files should not be opened for write access unless one of these operations must be performed. The protection provided to files opened for read access is independent of any particular operating system.

Only one array file may be opened for write access by any program at any one time. When a program attempts to open a file for write access, an error is signalled if another file is already open for write access, or if the file is already open for read access. An error is also signalled if a program attempts to open a file for read access if the file is already open for write access. (Errors are signalled through the standard SPICELIB error handling mechanism.)

Handles

When a file is opened for either kind of access, it is assigned an integer 'handle'. A mapping between handles and Fortran logical units is maintained by the DAF subroutines.

As a means of accessing files, handles have two advantages over logical unit numbers.

1. They reduce the possibility that two or more program units of the same program will interfere with each other when both need to access the same array file. When an array file is opened by a program for the first time, the file is connected to a logical unit, and the unit is mapped to a handle. If the program attempts to open the same file again, the handle is returned immediately, and a counter is incremented. The file is not disconnected from the logical unit until it has been closed as many times as it has been opened. (This is analagous to the creation of multiple links to a single file under the UNIX operating system.) Any one program unit is prevented from releasing a file that is still being used by other program units.

2. They allow the SPICELIB subroutines to prevent files opened for read access from being modified. (Positive handles are assigned to files opened for read access, negative handles to files opened for write access. Any attempt to modify a file with a positive handle signals an error.) Note that this scheme is independent of any file protection provided by the host operating system.

Subroutines

SPICELIB contains 38 subroutines that can be used to create, populate, and manipulate array files. The name of each routine begins with the letters 'DAF', followed by a two- or three-character mnemonic. For example, the routine that begins a forward search of an array file is named DAFBFS, pronounced 'DAF-B-F-S'. A complete list of mnemonics, translations, and calling sequences can be found at the end of this document.

Each subroutine is prefaced by a complete SPICELIB module header, which describes inputs, outputs, restrictions, and exceptions, discusses the context in which the subroutine should be used, and shows typical examples of its use. Any discussion of the subroutines in this article is intended as an introduction: the final documentation for any subroutine is its module header.

Whenever a subroutine appears in an example, the translation of the mnemonic part of its name will appear to the right of the reference, in braces. For example,

```
CALL DAFBFS ( HANDLE )                          { Begin forward search }
```

Examples will make use of the structured DO ... END DO and DO WHILE ... END DO statements supported by the VAX/VMS Fortran compiler. These statements are easily converted to the standard equivalents

```
        DO label var = e1, e2, e3
           stmt
label   CONTINUE
```

and

```
label   IF       ( expr )
       .THEN
           stmt
           GO TO label
        END IF
```

Opening, closing array files

An existing array file can be opened by supplying the name of the file to DAFOPR (for read access) or DAFOPW (for write access). Each routine returns a file handle, which must be used for all subsequent access to the file.

```
CALL DAFOPR ( FNAME, HANDLE )                   { Open for read }
CALL DAFOPW ( FNAME, HANDLE )                   { Open for write }
```

Once opened, an array file can be closed by supplying its handle to DAFCLS.

```
CALL DAFCLS ( HANDLE )                          { Close }
```

Creating array files

A new array file can be created by supplying the name of the file, values for ND and NI, an internal file name, and the number of records to be reserved.

```
CALL DAFOPN ( FNAME,                                    { Open new }
              ND,
              NI,
              IFNAME,
              RESV,
              HANDLE  )
```

The internal name of an array file is simply a string of up to 60 characters, which may be used to characterize the contents of the file. Its primary value is that, being internal to the file, it remains unchanged when the file is transferred between environments.

Any number of records may be reserved at the front of an array file. By definition, these records are invisible to DAF subroutines, and may contain any information that the user wishes to store in them.

Once created, a new array file remains open for write access until explicitly closed.

Adding arrays

A new array can be added to an existing array file by calling three routines: DAFBNA, DAFADA, and DAFENA.

DAFBNA requires the handle of the file (which must be open for write access), the array name, and the array summary.

```
CALL DAFBNA ( HANDLE, NAME, SUM )                { Begin new array }
```

The summary is packed by DAFPS, which requires two arrays containing the double precision and integer components of the summary. It also requires the values of ND and NI for the file.

```
CALL DAFPS ( DC, IC, ND, NI, SUM )                { Pack summary }
```

The final two integer components are always used to store the initial and final addresses of the array. They are filled in after the array has been stored: any values for these components supplied by the user are ignored.

The elements of the array are added by DAFADA. The elements may be supplied in one shot,

```
CALL DAFADA ( ELTS, N )                          { Add data to array }
```

or in any number of installments,

```
DO WHILE ( MORE )
   ...
   CALL DAFADA ( ELTS, N )                       { Add data to array }
END DO
```

Once the entire array has been supplied, DAFENA makes the addition permanent.

```
CALL DAFENA                                      { End new array }
```

If the process is aborted before DAFENA is called, the summary and name are not stored, and the new array does not become a permanent member of the file. Space allocated for elements of the array cannot be removed from the file; however, it will be overwritten by the elements of the next array added to the file.

Searching

The summaries and names for the arrays in a file can be examined without looking at the actual arrays.

Subroutines DAFBFS and DAFFNA are used to search an array file in forward order. DAFBFS places a pointer at the head of the doubly-linked list formed by the arrays in the file. Each call to DAFFNA moves the pointer to the next array in the list. (The first call to DAFFNA moves the pointer to the first array.) DAFFNA returns a logical flag which is true whenever another array has been found, and is false when the tail of the list has been reached. All forward searches are variations on the following template:

```
CALL DAFBFS ( HANDLE )                           { Begin forward search }
CALL DAFFNA ( FOUND  )                               { Find next array }

DO WHILE ( FOUND )
   ...
   CALL DAFFNA ( FOUND )                             { Find next array }
END DO
```

Subroutines DAFBBS and DAFFPA are likewise used to search an array file in backward order. DAFBBS moves the pointer to the tail (instead of the head) of the list; DAFFPA moves the pointer to the previous (instead of the next) array in the list. The template shown above is modified to conduct backward searches by replacing calls to DAFBFS and DAFFNA with calls to DAFBBS and DAFFPA, respectively:

```
CALL DAFBBS ( HANDLE )                          { Begin backward search }
CALL DAFFPA ( FOUND  )                            { Find previous array }

DO WHILE ( FOUND )
   ...
   CALL DAFFPA ( FOUND )                          { Find previous array }
END DO
```

Once a search has begun, the pointer may be moved in either direction.

After the pointer has been moved to a new array, the summary and name of the array are retrieved by DAFGS and DAFGN.

```
CALL DAFBBS ( HANDLE )                          { Begin backward search }
CALL DAFFPA ( FOUND  )                            { Find previous array }

DO WHILE ( FOUND )
   CALL DAFGS ( SUM  )                                  { Get summary }
   CALL DAFGN ( NAME )                                     { Get name }
   ...
   CALL DAFFNA ( FOUND )                          { Find next array }
END DO
```

Once returned, a name can be examined directly. However, a summary must first be unpacked into its components by subroutine DAFUS:

```
CALL DAFUS ( SUM, ND, NI, DC, IC )              { Unpack summary }
```

The name and summary are used to determine whether the current array is of interest. For example, if the arithmetic mean of the elements in each array of an array file is stored in the second double precision component of the summary, then the following code fragment determines the initial and final addresses (IA and FA) of the array with the greatest average. (Assume function DPMIN returns the smallest double precision number supported in the host environment.)

```
MAXAVG = DPMIN()

CALL DAFBFS ( HANDLE )                          { Begin forward search }
CALL DAFFNA ( FOUND  )                            { Find next array }
```

```
DO WHILE ( FOUND )
   CALL DAFGS ( SUM )                                    { Get summary }
   CALL DAFUS ( SUM, ND, NI, DC, IC )                    { Unpack summary }

   IF ( DC(2) .GT. MAXAVG ) THEN
      MAXAVG  = DC(2)
      IA      = IC(NI-1)
      FA      = IC(NI  )
   END IF

   CALL DAFFNA ( FOUND )                                 { Find next array }
END DO
```

Recall that the final two integer components of any array summary—IC(NI-1) and IC(NI)—contain the initial and final addresses of the array.

Accessing array elements

After an array of interest has been located, the entire array or any contiguous set of elements can be accessed—read or updated—by supplying a pair of addresses. Elements are read by DAFRDA and written by DAFWDA.

The following code fragment continues the example above by subtracting the average from each of the elements in the array. (Recall that IA and FA contain the initial and final addresses of the array.)

```
CALL DAFRDA ( HANDLE, IA, FA, DATA )              { Read data from address }

DO I = 1, FA - IA + 1
   DATA(I) = DATA(I) - MAXAVG
END DO

CALL DAFWDA ( HANDLE, IA, FA, DATA )              { Write data to address }
```

Note that it is not necessary to retrieve the entire array at once. The following code fragment performs the same operation on groups of CHUNK elements at a time.

```
FIRST = IA

DO WHILE ( FIRST .LE. FA )
   LAST  = MIN ( FA,     FIRST + CHUNK - 1 )
   CALL DAFRDA ( HANDLE, FIRST, LAST, DATA )   { Read data from address }

   DO I = 1, CHUNK
      DATA(I) = DATA(I) - MAXAVG
   END DO

   CALL DAFWDA ( HANDLE, FIRST, LAST, DATA )   { Write data to address }
   FIRST = FIRST + CHUNK
```

```
END DO
```

This technique, which allows an array of unknown size to be processed using a fixed amount of local storage, is useful when the arrays stored in an array file may be arbitrarily large.

Updating summaries, names

In the example, once the average value of the array has been subtracted from each element of the array, the value for the average stored in the summary is no longer valid (the average is now zero) and should be changed.

Subroutines DAFRS and DAFRN are analagous to subroutines DAFGS and DAFGN. DAFGS returns the summary for the array to which the pointer currently points; DAFRS replaces it. DAFGS returns the name of the array to which the pointer currently points; DAFRN replaces it.

If the index, K, of the updated array is known, then the new average for the array (zero) is stored by the following code fragment.

```
CALL DAFBFS ( HANDLE )                          { Begin forward search }

DO I = 1, K
   CALL DAFFNA ( FOUND  )                       { Find next array }
END DO

CALL DAFGS ( SUM )                                 { Get summary }
CALL DAFUS ( SUM, ND, NI, DC, IC )               { Unpack summary }

DC(2) = 0.DO

CALL DAFPS ( DC, IC, ND, NI, SUM )                 { Pack summary }
CALL DAFRS ( SUM )                               { Replace summary }
END DO
```

Buffering

Note that unless the value of CHUNK is 128 and the initial address of the array happens to correspond to the first word of a physical record (neither of which is very likely), each call to DAFRDA or DAFWDA will involve reading partial records. In general, successive calls will refer to different parts of at least one record.

In fact, records from array files are saved in a buffer as they are read. If any part of a record is needed, it can frequently be returned directly from the buffer, without accessing the file again. In particular, when an entire array is

accessed sequentially, as in the example above, each of the necessary records is read exactly one time. When the elements of an array are accessed more randomly, the number of file accesses may increase somewhat.

It is possible, at any point in a program, to determine the number of file accesses prevented by the buffering scheme. The subroutine DAFNRR returns the number of physical records actually read, and the number of records or partial records that have been requested, as illustrated below:

```
CALL DAFNRR ( READS, REQS )                    { Number of reads, requests }

RATIO = INT ( ( DBLE(READS) / DBLE(REQS) * 100.D0 )
WRITE (*,*) 'Reads/requests (%) = ', RATIO
```

Ideally, the ratio of reads to requests should approach zero. In the worst case, where it approaches one, the size of the buffer should probably be adjusted. (The module headers for DAFRDR and DAFWDR provide details on adjusting the buffer size.)

Conversion and transfer

In order to be transferred to a new environment, a binary array file must be converted to an equivalent text file containing only printable ASCII characters. In order to be used in the new environment, it must be reconverted to a binary file.

The simplest way to convert a binary array file to an equivalent text file is to supply the names of both files to DAFB2A, as in the following example.

```
CALL DAFB2A ( BINARY, ASCII )                  { Binary to ASCII }
```

Once transferred to the new environment, the text file is reconverted by DAFA2B.

```
CALL DAFA2B ( ASCII, BINARY, RESV )            { ASCII to binary }
```

Note that the user is required to specify the number of records to be reserved in the new binary file. Reserved records in the original binary file are ignored by DAFB2A.

When converting a binary file for transfer (or archival), it may be necessary to add additional information—catalog or history information, for example—to the resulting text file. Subroutine DAFB2T is a slightly more flexible version of DAFB2A. Where DAFB2A takes the name of a text file, DAFB2T takes

the logical unit number of a text file that has already been opened. DAFB2A closes the text file that it writes, but DAFB2T leaves the file open.

The corresponding routine, DAFT2B, also takes a logical unit instead of a file name, and also leaves the text file open. The following code fragment creates a text file containing an ASCII array file preceded and followed by standard markers.

```
OPEN ( FILE=ASCII, UNIT=TEXT, STATUS='NEW' )
WRITE (TEXT,*) '*** BEGIN ARRAY FILE ***'

CALL DAFB2T ( BINARY, TEXT )                    { Binary to text }

WRITE (TEXT,*) '*** END ARRAY FILE ***'
CLOSE ( TEXT )
```

The following code fragment reconverts the resulting text file into a binary array file.

```
OPEN ( FILE=ASCII, UNIT=TEXT, STATUS='OLD' )
READ (TEXT,FMT='(A)') BEGMRK

CALL DAFT2B ( TEXT, BINARY, RESV )              { Text to binary }

READ (TEXT,FMT='(A)') ENDMRK
CLOSE ( TEXT )
```

This technique can also be used to transfer the contents of reserved records and arrays in a single text file.

Structure

Every array file is a Fortran-77 direct access file, created by the following statement (or its equivalent):

```
OPEN ( UNIT   = unit,
       FILE   = file name,
       ACCESS = 'DIRECT',
       RECL   = record length,
       STATUS = 'NEW'          )
```

The record length is processor dependent. The smallest possible value should be selected such that each record in the file is large enough to contain 128 double precision numbers or 1000 characters. Some suitable values for several compilers are shown below.

Compiler	Record length
IBM/Lahey	1024
IBM/Microsoft	1024
Unisys/FTN	256
UNIX/f77	1024
VAX/Ultrix	1024
VAX/VMS	256

Organization

An array file contains five types of physical records:

1. A single 'file record'. This contains global information about the file.
2. An arbitrary number of 'reserved records'. By definition, these are invisible to DAF subroutines. The contents and formats of reserved records are left entirely to the user. The initial reserved record is always located in the second record of the file. The final reserved record immediately precedes the initial summary record (see below) of the file.
3. Some number of 'summary records'. These contain array summaries and pointers to other summary records. The number of summary records in a particular array file is a function of the number of arrays stored in the file.
4. Some number of 'name records'. These contain array names. An array file contains one name record for each summary record.
5. An arbitrary number of 'element records'. These contain elements of the arrays stored in the array file.

The file record

The file record is always the first physical record in an array file. It contains seven items.

1. A keyword ('NAIF/DAF'). This is used by the SPICELIB subroutines to verify that a particular file is in fact an array file, and not merely a direct access file with the same record length. When an array file is opened, an error is signalled if this keyword is not present.
2. The value of ND, the number of double precision components in each array summary.
3. The value of NI, the number of integer components in each array summary.

4. The internal name (60 characters) of the array file.
5. The record number of the initial summary record in the file.
6. The record number of the final summary record in the file.
7. The first free address in the file. This is the address at which the first element of the next array to be added to the file should be stored.

Summary records

The first three (double precision) words of each summary record are reserved for the following control information:

1. The record number of the next summary record in the file. (Zero if this is the final summary record.)
2. The record number of the previous summary record in the file. (Zero if this is the initial summary record.)
3. The number of summaries stored in this record.

The record pointers form the basis of the array list. Each summary record is linked to two other summary records, allowing the summaries to be retrieved in forward or backward order. (The links between adjacent summaries in a summary record are implicit.) The names can be retrieved from the corresponding name records. And the locations (initial and final addresses) of the arrays themselves are stored in the summaries.

Although the control items are integer values, they are stored as double precision numbers. This allows summary records and element records, which contain only double precision numbers, to be buffered using the same mechanism.

The control items are followed immediately by the summaries themselves. If NS is the number of elements in each summary array, then the first summary is stored in words 4 through NS+3; the second summary is stored in words NS+4 through 2(NS)+3; and so on.

The number of summaries that can fit in a single summary record is a function of ND and NI (It is the largest integer, K, such that K(NS) is not greater than 125.) Unlike arrays, summaries are never split across physical record boundaries, so the end of each summary record may remain unused.

Whenever the number of summaries stored in the final summary record reaches the maximum number that will fit, a new (empty) summary record is added to the end of the file.

Name records

Each name record contains nothing but array names. A new name record is added to the file each time a new summary record is added, in the record immediately following the summary record.

If NC is the number of characters in each name, the first name is stored in characters 1 through NC of the record; the second name is stored in characters NC+1 through 2(NC); and so on.

Element records

Most of the records in any array file are element records. Element records hold the elements of the arrays stored in the file. (The other records are used for accounting purposes only.)

Each element record contains up to 128 double precision numbers. An element record is always full (contains 128 numbers) unless it immediately precedes a summary record, in which case it may be partially filled.

The elements stored in a particular element record may belong to more than one array. However, if two elements in an element record belong to the same array, they are always adjacent within that record.

A particular element record always lies between two summary records, or between a summary record and the end of the file. In either case, the summaries for the arrays whose elements are contained in the record are always stored in the nearest preceding summary record. (The array names are stored in the associated name record.)

An example

The following example illustrates the use of addresses and lists within an array file by showing how a simple array file might be created, and how arrays might be added to that file.

Throughout the example, the following notations will be used:

- Within the file record, ND and NI are the values of the parameters that define the format of the file; RI and RF are the record numbers of the initial and final summary records in the file; and FAA is the first free address in the file.
- Within a particular summary record, NEXT and PREV are the record numbers of the next and previous summary records in the file; and NSUM is the number of summaries stored in the record.

The first step in creating a file is to select values for ND and NI. Normally, these are relatively small, allowing several summaries to fit in each summary record, and increasing the speed with which the file can be searched. The example will be easier to follow, however, if the number of summaries that can fit in a summary record is minimized. Therefore, in this example ND and NI will take on unusually large values:

```
ND = 25
NI = 27
```

Each array summary requires 39 double precision words of storage, so each summary record can hold 3 summaries. If 'Summary(i)[j]' represents the j'th element of the i'th summary array, then the layout of a typical summary record is shown below.

Word	Value
1	NEXT
2	PREV
3	NSUM
4	Summary(1)[1]
5	Summary(1)[2]
	...
42	Summary(1)[39]
43	Summary(2)[1]
	...
81	Summary(2)[39]
82	Summary(3)[1]
	...
110	Summary(3)[39]
111	Unused
	...
128	Unused

Each array name requires 312 characters of storage, so each name record can hold 3 names. If 'Name(i)[j]' represents the j'th character of the i'th name, then the layout of a typical summary record is shown below.

Character	Value
1	Name(1)[1]
2	Name(1)[2]
	...
312	Name(1)[312]
313	Name(2)[1]
	...
624	Name(2)[312]
625	Name(3)[1]
	...
936	Name(3)[312]
937	Unused

```
                ...
1000        Unused
```

Assume that ten records are reserved. These are stored in records 2-11, so the initial summary record is stored in record 12. Because the file is empty, the initial summary record is also the final summary record.

```
RI = 12
RF = 12
```

The lone name record for the file is stored immediately after the summary record, in record 13. Therefore the first first free address, FFA, in the file is the first word in record 14:

```
FFA = word + (record - 1) * 128

    =    1 + (14     - 1) * 128

    = 1664
```

Except for the internal file name, which can be ignored, the information required to create the file record is complete. For the rest of the example, the file record will be depicted as a collection of values enclosed by braces and preceded by a record number:

```
r   { ND=a, NI=b, RI=c, RF=d, FFA=e }
```

Thus, the file record for this example file is initially

```
1   { ND=3, NI=5, RI=12, RF=12, FFA=1664 }
```

Because there is only one summary record, the values of NEXT and PREV in that record are both zero. Because the file contains no arrays, the value of NSUM is also zero. The information needed to create the summary record is complete. For the rest of the example, each summary record will be depicted as a collection of values enclosed by angle brackets and preceded by a record number,

```
r   < NEXT=a, PREV=b, NSUM=c, (d,e), (f,g), (h,i) >
```

The ordered pairs enclosed in parentheses are the initial and final addresses of the arrays whose summaries are contained in the record. The remaining

components of each summary are ignored. Thus, the lone summary record for this example file is initially

```
12   < NEXT=0, PREV=0, NSUM=0, (0,0), (0,0), (0,0) >
```

Name records will always be depicted as

```
r   < " " >
```

Data records will always be depicted as

```
r   < N >
```

where N is the number of elements stored in the record.

Once the initial summary and name records have been written, the file is complete, if uninteresting:

```
 1   { ND=3, NI=5, RI=12, RF=12, FFA=1664 }
       ...
12   < NEXT=0, PREV=0, NSUM=0, (0,0), (0,0), (0,0) >
13   < " " >
```

Assume that an array containing 100 elements is to be added to the file. The array must be stored contiguously, beginning at the first free address. Thus, its initial and final addresses will be 1664 and 1763, respectively. The entire array fits into a single record, so one data record must be added to the file. The value of NSUM in the summary record must be incremented by one. The new value of FFA is the address following the final address of the new array: 1764. This must be stored in the file record.

```
 1   { ND=3, NI=5, RI=12, RF=12, FFA=1764 }
       ...
12   < NEXT=0, PREV=0, NSUM=1, (1664,1763), (0,0), (0,0) >
13   < " " >
14   < 100 >
```

Assume that a second array, containing 200 elements, is to be added to the file. The elements must be stored between addresses 1764 and 1963. The array will fill the remainder of the first data record, all of a second record, and part of a third, so two data records must be added to the file. The value of NSUM is in the summary record must be incremented again. And the new value of FFA (1964) must be stored in the file record.

```
 1   { ND=3, NI=5, RI=12, RF=12, FFA=1964 }
```

```
    ...
12  < NEXT=0, PREV=0, NSUM=2, (1664,1763), (1764,1963), (0,0) >
13  < " " >
14  < 128 >
15  < 128 >
16  <  44 >
```

To add a third array, containing 150 elements, the process is repeated. The elements must be stored between addresses 1964 and 2113. The array will fill the remainder of the third data record, and part of a fourth, so one new data record must be added. The value of NSUM is in the summary record must be incremented again. And the new value of FFA (2114) must be stored in the file record.

```
 1  { ND=3, NI=5, RI=12, RF=12, FFA=2114 }
    ...
12  < NEXT=0, PREV=0, NSUM=3, (1664,1763), (1764,1963), (1964,2113) >
13  < " " >
14  < 128 >
15  < 128 >
16  < 128 >
17  <  66 >
```

Note that the final summary record is full, so new summary and name records must be added to the file. (Record 17 will remain only partially filled.) The values of NEXT and PREV in the summary records must be adjusted so that the records point to each other:

```
 1  { ND=3, NI=5, RI=12, RF=12, FFA=2114 }
    ...
12  < NEXT=18, PREV=0, NSUM=3, (1664,1763), (1764,1963), (1964,2113) >
13  < " " >
14  < 128 >
15  < 128 >
16  < 128 >
17  <  66 >
18  < NEXT=0, PREV=12, NSUM=0, (0,0), (0,0), (0,0) >
19  < " " >
```

The value of RF in the file record must point to the new summary record:

```
 1  { ND=3, NI=5, RI=12, RF=18, FFA=2114 }
    ...
12  < NEXT=18, PREV=0, NSUM=3, (1664,1763), (1764,1963), (1964,2113) >
13  < " " >
14  < 128 >
15  < 128 >
16  < 128 >
17  <  66 >
18  < NEXT=0, PREV=12, NSUM=0, (0,0), (0,0) >
19  < " " >
```

And the value of FFA in the file record must point to the first word in the first record following the new name record (address 2432):

```
 1   { ND=3, NI=5, RI=12, RF=18, FFA=2432 }
       ...
12   < NEXT=18, PREV=0, NSUM=3, (1664,1763), (1764,1963), (1964,2113) >
13   < " " >
14   < 128 >
15   < 128 >
16   < 128 >
17   <  66 >
18   < NEXT=0, PREV=12, NSUM=0, (0,0), (0,0) >
19   < " " >
```

Adding a fourth or fifth array is just like adding the first: the necessary data records are added; the summary and name records are updated; and the value of FFA is replaced.

Adding a sixth array is just like adding the third: the necessary data records are added; the summary and name records are updated; new summary and name records are added; and the values of RF and FFA are replaced.

Text format

In order to be transferred over an eletronic network to a new environment, a binary array file is normally converted to an equivalent text file.

Numbers are always converted to equivalent text representations (for example, '0.312948596339236E+03' or '17'). Strings are always enclosed in apostrophes,

```
'Surface features'
```

making them suitable for Fortran list-directed input.

Blank lines within the text file are not significant.

The first line of the text file is always the keyword 'NAIF/DAF'. The second line always contains the values of ND and NI for the file.

The third line of the text file is the internal name of the array file.

Each array is preceded by a one ('1'), on a line by itself, indicating that an array is to follow. The final array is followed by a zero ('0'), also on a line by itself, indicating that no arrays remain.

The first line of each array is the name of the array.

The next few lines contain the double precision and integer components of the array summary. They may be arranged on any number of lines, so long as they appear in the proper order.

The elements of the array are printed in groups of arbitrary size. Each group is preceded by the number of elements in the group, on a line by itself. The elements in the group may be arranged on any number of lines, solong as they appear in the proper order. The final group is followed by a zero ('0'), on a line by itself, indicating that no elements remain.

The final line of each array is the name of the array, repeated as a consistency check.

The final line of the file is the internal file name of the array file, repeated as a consistency check.

(Subroutines DAFA2B and DAFT2B signal errors whenever these checks fail.)

Example 1

The next several sections present example programs and subroutines to show how the DAF subroutines can be used to manipulate array files.

All subroutines and functions used in the examples are from SPICELIB. The convention of expanding DAF subroutine names will be dropped for these examples.

The first example is a complete program to convert a binary array file (DAF) to an equivalent text file, suitable for transfer to a new environment. The program queries the user for the names of the binary file to be converted and the text file to be created.

```
PROGRAM B2T

CHARACTER*(128)        BINARY
CHARACTER*(128)        TEXT

WRITE (*,*)           'Name of binary file?'
READ  (*,FMT='(A)')    BINARY

WRITE (*,*)           'Name of text file?'
READ  (*,FMT='(A)')    TEXT

CALL DAFB2A ( BINARY, TEXT )

END
```

Converting from text back to binary is just as simple:

```
PROGRAM T2B

CHARACTER*(128)        BINARY
CHARACTER*(128)        TEXT

WRITE (*,*)           'Name of text file?'
READ  (*,FMT='(A)')    TEXT

WRITE (*,*)           'Name of binary file?'
READ  (*,FMT='(A)')    BINARY


CALL DAFA2B ( TEXT, BINARY, 0 )

END
```

The subroutines DAFB2A and DAFA2B can just as easily be embedded in programs with more sophisticated (but less portable) interfaces—using X Windows, the Macintosh toolkit, and so on.

Example 2

The next example is a subroutine, SUMARR, which summarizes the contents of an array in an arbitrary array file. It assumes that a search (forward or backward) is in progress, and that an array has been found.

The subroutine takes two arrays as inputs, containing 'print names' for the double precision and integer components of the summary. The value of each component is preceded by this name.

Function LASTNB returns the index of the last non-blank character in a string.

```
      SUBROUTINE SUMARR ( DNAMES, INAMES )

      CHARACTER*(*)       DNAMES ( * )
      CHARACTER*(*)       INAMES ( * )

C
C     SPICELIB functions
C
      INTEGER             LASTNB

C
C     Local variables
C
      CHARACTER*(80)      PNAME

      DOUBLE PRECISION    DC     ( 125 )
      DOUBLE PRECISION    SUM    ( 125 )

      INTEGER             HANDLE
      INTEGER             I
      INTEGER             IC     ( 250 )
      INTEGER             ND
      INTEGER             NI
      INTEGER             LONG

C
C     Look up the handle of the file, and the summary of the
C     array most recently found.
C
      CALL DAFGH  ( HANDLE          )
      CALL DAFHSF ( HANDLE, ND, NI )

C
C     Get, and unpack, the summary of the array. Note that SUM,
C     DC, and IC are dimensioned large enough to hold the biggest
C     possible summary.
C
      CALL DAFGS  ( SUM )
      CALL DAFUS  ( SUM, ND, NI, DC, IC )

C
C     Find the length of the longest print name. All names will
C     be transferred into a temporary string for printing.
```

```
C     Shorter names will be followed by enough blanks to make
C     the components line up.
C
      LONG = 1

      DO I = 1, ND
         LONG = MAX ( LONG, LASTNB ( DNAMES(I) ) )
      END DO

      DO I = 1, NI
         LONG = MAX ( LONG, LASTNB ( INAMES(I) ) )
      END DO

C
C     Write the summary: an aligned list of components, each
C     preceded by a descriptive name.
C
      WRITE (*,*)
      WRITE (*,*) 'Summary'
      WRITE (*,*) '-------'
      WRITE (*,*)

      DO I = 1, ND
         PNAME = DNAMES(I)
         WRITE (*,*) PNAME(1:LONG), ' :', DC(I)
      END DO

      DO I = 1, NI
         PNAME = INAMES(I)
         WRITE (*,*) PNAME(1:LONG), ' :', IC(I)
      END DO

      RETURN
      END
```

The following program uses SUMARR to summarize all of the arrays in a specific kind of array file. Each summary in the file contains four double precision components (minimum, maximum, average, standard deviation) and three integer components (sort flag, initial address, final address).

Note that the program conforms to the template for forward searches described earlier.

```
      PROGRAM SUMDAF

      CHARACTER*(40)    DNAMES  ( 4 )
      CHARACTER*(128)   FILE
      CHARACTER*(40)    INAMES  ( 3 )

      INTEGER           HANDLE

      LOGICAL           FOUND

      DATA              DNAMES     / 'Largest value',
                                     'Smallest value',
                                     'Average value',
```

```
.                                   'Standard deviation'   /

 DATA              INAMES      / 'Sorted (1=yes, 0=no)',
.                                  'Begins at address',
.                                  'Ends at address'      /

 WRITE (*,*)
 WRITE (*,*)          'Name of file?'
 READ  (*,FMT='(A)')  FILE

 CALL DAFOPR ( FILE, HANDLE )

 CALL DAFBFS ( HANDLE )
 CALL DAFFNA ( FOUND  )

 DO WHILE ( FOUND )
    CALL SUMARR ( DNAMES, INAMES )
    CALL DAFFNA ( FOUND )
 END DO

 END
```

The summary of a typical array is shown below:

```
Summary
-------

Largest value         : 221.42123212345
Smallest value        : 17.332467369560
Average value         : 148.37378239493
Standard deviation    : 21.263546586965
Sorted (1=yes, 0=no)  : 0
Begins at address     : 21463
Ends at address       : 29271
```

Example 3

The next example is a subroutine to copy an entire array from one array file to another. It assumes that a search (forward or backward) is in progress, and that an array has been found.

It takes a single input: the name of the file to which the array is to be copied.

```
      SUBROUTINE COPYA ( FILE )

      CHARACTER*(*)       FILE

C
C     Local variables
C
      CHARACTER*(1000)    NAME

      INTEGER             FA
      INTEGER             FIRST
      INTEGER             FROM
      INTEGER             HANDLE
      INTEGER             IA
      INTEGER             IC        ( 250 )
      INTEGER             LAST
      INTEGER             ND
      INTEGER             NI
      INTEGER             TO

      DOUBLE PRECISION    DATA      ( 100 )
      DOUBLE PRECISION    DC        ( 125 )
      DOUBLE PRECISION    SUM       ( 250 )

C
C     Get the handle of the source file, and the values of
C     ND and NI for that file.
C
      CALL DAFGH  ( FROM )
      CALL DAFHSF ( FROM, ND, NI )

C
C     Open the target file for write access.
C
      CALL DAFOPW ( FILE, TO )

C
C     Get the summary and name for the array to be copied.
C     Start the new array.
C
      CALL DAFGS ( SUM  )
      CALL DAFGN ( NAME )

      CALL DAFBNA ( TO, NAME, SUM )

C
C     Unpack the summary to get the initial and final addresses
C     of the original array.
```

```
C
      CALL DAFUS ( SUM, ND, NI, DC, IC )
      IA = IC(NI-1)
      FA = IC(NI  )

C
C     Copy the elements in groups of 100.
C
      FIRST = IA

      DO WHILE ( FIRST .LE. FA )
         LAST  = MIN ( FA, FIRST + 100 - 1 )
         CALL DAFRDA ( HANDLE, FIRST, LAST, DATA )

         CALL DAFADA ( DATA, LAST - FIRST + 1 )
         FIRST = FIRST + 100
      END DO

C
C     Make the addition permanent, then close the target file.
C
      CALL DAFENA
      CALL DAFCLS ( TO )

      RETURN
      END
```

Example 4

The final example is a complete program to copy the arrays in one file to a second file, so that the arrays in the new file are sorted according to the value of the first double precision component of each summary.

The program assumes that the file contains fewer than 1000 arrays.

Subroutine ORDERD creates an order vector for a double precision array. Subroutine COPYA, defined in the previous example, is used to copy the arrays.

```
      PROGRAM DAFSRT

      CHARACTER*(128)       SOURCE
      CHARACTER*(128)       TARGET

      DOUBLE PRECISION      DC      (  125 )
      DOUBLE PRECISION      SUM     (  125 )
      DOUBLE PRECISION      VALUE   ( 1000 )

      INTEGER               HANDLE
      INTEGER               IC      (  250 )
      INTEGER               NA
      INTEGER               ND
      INTEGER               NI
      INTEGER               ORDER   ( 1000 )
      INTEGER               TO

      LOGICAL               FOUND

C
C     Prompt for the names of the source and target files.
C
      WRITE (*,*)
      WRITE (*,*)          'Name of source (unsorted) file?'
      READ  (*,FMT='(A)')  SOURCE

      WRITE (*,*)
      WRITE (*,*)          'Name of target (sorted) file?'
      READ  (*,FMT='(A)')  TARGET

C
C     Open the source file, and look up the values of ND and NI for
C     the file.
C
      CALL DAFOPR ( SOURCE, HANDLE         )
      CALL DAFHSF (         HANDLE, ND, NI )

C
C     Collect the values of the first double precision component
C     of each summary.
C
      CALL DAFBFS ( HANDLE )
      CALL DAFFNA ( FOUND  )
```

```
      NA = 0
      DO WHILE ( FOUND )
         CALL DAFGS ( SUM )
         CALL DAFUS ( SUM, ND, NI, DC, IC )

         NA        = NA + 1
         VALUE(NA) = DC(1)

         CALL DAFFNA ( FOUND )
      END DO

C
C     Create an order vector for the values, such that ORDER(1)
C     is the index of the array with the smallest value, ORDER(2)
C     is the index of the array with the next smallest value, and
C     so on.
C
      CALL ORDERD ( VALUES, NA, ORDER )

C
C     Open the new file, then close it immediately. COPYA will
C     reopen it as each array is copied.
C
      CALL DAFOPW ( TARGET, ND, NI, 'Sorted file', 0, I )
      CALL DAFCLS (                                   I )

C
C     Look up the arrays in the specified order (starting from the
C     beginning of the file each time). Copy each one as it is found
C     to the target file.
C
      DO I = 1, NA
         CALL DAFBFS ( HANDLE )

         DO J = 1, ORDER(I)
            CALL DAFFNA ( FOUND )
         END DO

         CALL COPYA ( TARGET )
      END DO

      END
```

Summary of mnemonics

SPICELIB contains 38 subroutines that can be used to create, populate, and manipulate array files. The name of each routine begins with the letters 'DAF', followed by a two- or three-character mnemonic. For example, the routine that begins a forward search of an array file is named DAFBFS, pronounced 'DAF-B-F-S'. The following is a complete list of mnemonics and translations, in alphabetical order.

A2B	**ASCII to binary**
ADA	**Add data to array**
ARW	**Address to record/word**
B2A	**Binary to ASCII**
B2T	**Binary to text**
BBS	**Begin backward search**
BFS	**Begin forward search**
BNA	**Begin new array**
CLS	**Close**
ENA	**End new array**
FNH	**File name to handle**
FNA	**Find next array**
FPA	**Find previous array**
GH	**Get handle**
GN	**Get name**
GS	**Get summary**
HFN	**Handle to file name**
HSF	**Handle to summary format**
HLU	**Handle to logical unit**
LUH	**Logical unit to handle**
NRR	**Number of reads, requests**
OPN	**Open new**
OPR	**Open for read**
OPW	**Open for write**
PS	**Pack summary**
RCR	**Read character record**
RDA	**Read data from address**
RDR	**Read precision record**
RFR	**Read file record**
RN	**Replace name**
RS	**Replace summary**
RWA	**Record/word to address**
T2B	**Text to binary**
US	**Unpack summary**
WCR	**Write character record**
WDA	**Write data to address**
WDR	**Write double precision record**
WFR	**Write file record**

Many of the subroutines listed here are not normally used except to support other subroutines. For example, subroutines that read and write records (RCR, RDR, RFR, WCR, WDR, WFR) should not normally be called by a typical user.

Summary of calling sequences

The calling sequences for the DAF subroutines are summarized below. Subroutines are grouped by function.

Opening, closing files:

```
OPR ( FNAME,                          HANDLE )
OPW ( FNAME,                          HANDLE )
OPN ( FNAME, ND, NI, IFNAME, RESV, HANDLE )
CLS (                                 HANDLE )
```

Adding an array to a file:

```
BNA ( HANDLE, NAME, SUM )
ADA ( DATA,   N         )
ENA
```

Finding an array within in a file:

```
BFS ( HANDLE )
FNA ( FOUND  )

BBS ( HANDLE )
FPA ( FOUND  )

GH ( HANDLE )
GN ( NAME   )
GS ( SUM    )
RN ( NAME   )
RS ( SUM    )

US  ( SUM, ND, NI, DC, IC      )
PS  (      ND, NI, DC, IC, SUM )
```

Reading and writing arrays:

```
RDA ( HANDLE, BEGIN, END, DATA )
WDA ( HANDLE, BEGIN, END, DATA )
```

Converting array files:

```
B2A ( BINARY, ASCII        )
A2B ( ASCII,  BINARY, RESV )

B2T ( BINARY, TEXT         )
T2B ( TEXT,   BINARY, RESV )
```

Reading, writing physical records:

```
RFR ( HANDLE, ND, NI, IFNAME, FWARD, BWARD, FREE )
WFR ( HANDLE, ND, NI, IFNAME, FWARD, BWARD, FREE )

RCR ( HANDLE, RECNO, CREC )
WCR ( HANDLE, RECNO, CREC )

RDR ( HANDLE, RECNO, BEGIN, END, DREC, FOUND )
WDR ( HANDLE, RECNO,             DREC        )
NRR ( READS,  REQS                           )
```

Internal conversions:

```
HSF ( HANDLE, ND, NI )

FNH ( FNAME,  HANDLE )
HFN ( HANDLE, FNAME  )
HLU ( HANDLE, UNIT   )
LUH ( UNIT,   HANDLE )

ARW ( ADDR, REC, WORD       )
RWA (       REC, WORD, ADDR )
```

Appendix C

SPACIT User's Guide

SPACIT Version 1.0 User's Guide

Navigation Ancillary Information Facility

Prepared by
C. H. Acton

NAIF Document No. 169.0
12 January 1990

Table of Contents

SPACIT (Version 1.0) User's Guide

Introduction

The SPACIT (S, P And C Kernel Information and conversion Tool) utility program, which is allied with the *SPICELIB* portable "toolkit," provides file conversion (text-to-binary and binary-to-text) and file summary functions for the SPK and CK SPICE file types.

SPK files are the physical realization of the SPICE system's logical S (spacecraft ephemeris) and P (planet, satellite, comet, and asteroid ephemeris) kernels. CK files are the physical realization of the SPICE system's logical C kernel, which contains data providing orientation (generally called pointing) for a spacecraft structure or science instrument.

Intended audience

Most scientists, engineers, or technical staff using SPICE system SPK or CK files will need to use the SPACIT utility program, and thus should read this user's guide.

References

Detailed specifications for NAIF's SPK and CK kernel files may be found in the following two NAIF documents.

1. "S- and P- Kernel (SPK) Specification and User's Guide"; NAIF Document No. 168; I. M. Underwood.
2. "C-Kernel Design Specification and User's Guide"; NAIF Document No. DRAFT dated 28 November 1989; R. E. Thurman.
3. "An Introduction to SPICELIB, Vols. I and II, Revision 1"; NAIF Document No. 121.1; 24 October 1988; edited by Chuck Acton.

SPACIT functions

The SPACIT utility provides four functions.

1. Convert a text format SPK or CK file to an equivalent binary format suitable for access using FORTRAN 77 subroutines ("readers") provided in SPICELIB, the NAIF software "toolkit" [1][2][3].
2. Convert a binary SPK or CK file to an equivalent text format suitable for porting to a different CPU or operating system environment.
3. Summarize the contents of a binary SPK file.
4. Summarize the contents of a binary CK file.

These functions are described below.

Using SPACIT

SPACIT is an interactive program, which prompts the user for whatever command directives and file names are needed to execute any of the functions listed above. Results of executing SPACIT are also displayed on the user's terminal.

The description of each SPACIT function shown below is accompanied by a sample dialogue. In these dialogues, outputs from the program are displayed in `typewriter` font, while inputs from the user are displayed in *italics.*

The syntax for executing the program under the Digital Equipment Corporation (DEC) VMS operating system is

```
$ run SPACIT
```

The syntax for the equivalent command may differ for other operating systems. The dialogues also use VMS syntax for file names:

```
diskname:[directory tree]filename.extension
```

Other systems will use other conventions.

Conversion of a Text Format SPK or CK File to Binary Format

SPK and CK files made by JPL flight project teams or NAIF for subsequent distribution to science or engineering teams, or to the Planetary Data System (PDS), are usually produced in text format to facilitate porting between heterogenous CPU/operating system environments. (The "penalty" paid for this approach to portability is an expansion in file size by about a factor of three.)

Text format SPK and CK files may be preceded and followed by labels used for identification and cataloging purposes by JPL's Space Flight Operations Center (SFOC) or the PDS. In this case, the files are said to be structured as Standard Format Data Units (SFDU). The current version (1.0) of SPACIT is unable to remove or bypass these labels. Hence, *the user must provide a means to strip all such SFDU labels from an SPK or CK text file before using SPACIT.* (SFOC or PDS may provide a tool to accomplish this task. Also, an expected augmentation of the SFDU methodology should allow future versions of SPACIT to bypass the labels.)

Once the text format SPK or CK file has been liberated from surrounding SFDU structures, if any, you can proceed with the conversion using SPACIT. The program requires the user to supply two file names: the name of the text format SPK or CK file to be converted, and the name of the equivalent binary file to be created, as illustrated by the following dialogue. The following example happens to be for an SPK file; operation on CK files is identical.

```
$ run SPACIT

------------------------------------------------------------

                    Welcome to SPACIT

          S, P, and C-kernel Information and Conversion Tool

------------------------------------------------------------


 1) Quit.
 2) Convert a text SPK or CK file to binary.
 3) Convert a binary SPK or CK file to ASCII.
 4) Summarize all or part of a binary SPK file.
 5) Summarize all or part of a binary CK file.

 2

 Enter the name of the input text file:

 SPICEDISK:[SPKFILES]SAR006.TSP

 Select a name for the output binary file:

 SPICEDISK:[SPKFILES]SAR006.BSP
```

```
Working ... please wait.

Binary file created: SPICEDISK:[SPKFILES]SAR006.BSP

Select a task. (Enter the appropriate number, followed by <Return>.)

 1) Quit.
 2) Convert a text SPK or CK file to binary.
 3) Convert a binary SPK or CK file to text.
 4) Summarize all or part of a binary SPK file.
 5) Summarize all or part of a binary CK file.


1
```

Conversion of a Binary SPK or CK File to Text Format

The conversion from binary format to text format differs from the conversion from text to binary in the following ways:

1. It is initiated by selecting option 3 instead of option 2 from the initial command menu.
2. The program prompts first for the name of a binary file (instead of a text file) to be converted.
3. The program prompts last for the name of a text file (instead of a binary file) to be created.

Compare the following dialogue with the one in the previous section.

```
$ run SPACIT

------------------------------------------------------------

                    Welcome to SPACIT

        S, P, and C-kernel Information and Conversion Tool

------------------------------------------------------------

Select a task. (Enter the appropriate number, followed by <Return>.)

 1) Quit.
 2) Convert a text SPK or CK file to binary.
 3) Convert a binary SPK or CK file to ASCII.
 4) Summarize all or part of a binary SPK file.
 5) Summarize all or part of a binary CK file.


3


Enter the name of the input binary file:

SPICEDISK:[SPKFILES]SAR006.BSP
```

```
Select a name for the output text file:

SPICEDISK:[SPKFILES]SAR006.TSP

Working ... please wait.

Text file created: SPICEDISK:[SPKFILES]SAR006.TSP

Select a task. (Enter the appropriate number, followed by <Return>.)

 1) Quit.
 2) Convert a text SPK or CK file to binary.
 3) Convert a binary SPK or CK file to text.
 4) Summarize all or part of a binary SPK file.
 5) Summarize all or part of a binary CK file.

1
```

Summarizing All or Part of a Binary SPK File

You can use SPACIT to display a summary of the segments included in an SPK file on your terminal screen. You may elect to see a summary of:

- every segment in a given file
- every segment containing data for a specified body
- every segment containing data for a specified time or time span

In addition to specifying the name of the SPK file to be summarized, you must also specify the name of a utility file, provided with the NAIF toolkit, which contains a tabulation of leapseconds needed for conversions between ephemeris time (ET) and Universal Time Coordinated (UTC, which is also frequently called Spacecraft Event Time, or SCET). This leapseconds kernel file is needed because time tags within an SPK file are in ET whereas SPACIT output uses the UTC times with which we are most familiar.

The following dialogue illustrates the use of SPACIT to summarize all of the segments in a particular SPK file.

```
$ run SPACIT

------------------------------------------------------------

                    Welcome to SPACIT

        S, P, and C-kernel Information and Conversion Tool

------------------------------------------------------------

Select a task. (Enter the appropriate number, followed by <Return>.)

 1) Quit.
```

```
 2) Convert a text SPK or CK file to binary.
 3) Convert a binary SPK or CK file to text.
 4) Summarize all or part of a binary SPK file.
 5) Summarize all or part of a binary CK file.
```

4

```
Enter the name of the file to be summarized:
```

SPICEDISK:[SPKFILES]SAR006.BSP

```
Enter name of the leapseconds kernel file (for time conversions):
```

SPICEDISK:[MISCKERNELS.LEAPSECONDS]LEAPSECONDS.KER

```
Choose one. (Enter the number.)

 1) Return to previous menu.
 2) Summarize entire file.
 3) Summarize segments for a particular body.
 4) Summarize segments for a given time or time span.
```

2

```
-------------------------------------------------------------------------
Segment identifier: Magellan ID RET 8/18/89
Body         : -18                          Center      : 2
From         : 1990 AUG 14 18:00:00.000
To           : 1990 AUG 14 21:10:00.000
Reference    : DE-200                       SPK Type    :1
-------------------------------------------------------------------------
-------------------------------------------------------------------------
Segment identifier: Magellan ID RET 8/18/89
Body         : 10                           Center      : 0
From         : 1990 AUG 14 18:00:00.000
To           : 1990 AUG 14 21:10:00.000
Reference    : DE-130                       SPK Type    :2
-------------------------------------------------------------------------
-------------------------------------------------------------------------
Segment identifier: Magellan ID RET 8/18/89
Body         : 2                            Center      : 0
From         : 1990 AUG 14 18:00:00.000
To           : 1990 AUG 14 21:10:00.000
Reference    : DE-130                       SPK Type    :2
-------------------------------------------------------------------------
-------------------------------------------------------------------------
Segment identifier: Magellan ID RET 8/18/89
Body         : 3                            Center      : 0
From         : 1990 AUG 14 18:00:00.000
To           : 1990 AUG 14 21:10:00.000
Reference    : DE-130                       SPK Type    :2
-------------------------------------------------------------------------
-------------------------------------------------------------------------
Segment identifier: Magellan ID RET 8/18/89
Body         : 299                          Center      : 2
From         : 1990 AUG 14 18:00:00.000
To           : 1990 AUG 14 21:10:00.000
```

```
Reference   : DE-130                          SPK Type    :2
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
Segment identifier: Magellan ID RET 8/18/89
Body        : 399                            Center      : 3
From        : 1990 AUG 14 18:00:00.000
To          : 1990 AUG 14 21:10:00.000
Reference   : DE-130                         SPK Type    :2
-------------------------------------------------------------------------------

Choose one. (Enter the number.)

 1) Return to previous menu.
 2) Summarize entire file.
 3) Summarize segments for a particular body.
 4) Summarize segments for a given time or time span.

1


Select a task. (Enter the appropriate number, followed by <Return>.)

 1) Quit.
 2) Convert a text SPK or CK file to binary.
 3) Convert a binary SPK or CK file to text.
 4) Summarize all or part of a binary SPK file.
 5) Summarize all or part of a binary CK file.

1
```

Had you chosen option 3 (Summarize segments for a particular body) or option 4 (Summarize segments for a specified time span), you would have also been prompted for the NAIF body ID, or for the time (a single epoch) or time span (both beginning and ending epochs). These options are illustrated for a CK file summary in the next section.

Summarizing All or Part of a Binary CK File

You can use SPACIT to display a summary of the segments included in a CK file on your terminal screen. You may elect to see a summary of:

- every segment in a given file
- every segment containing pointing data for a specified instrument
- every segment containing pointing data for a specified time or time span, using epochs specified in UTC or SCET
- every segment containing pointing data for a specified time or time span, using epochs specified in SCLK (spacecraft clock count)

In addition to specifying the name of the CK file to be summarized, you must also specify the names of two utility files provided with the NAIF toolkit. One contains a tabulation of leapseconds needed for conversions between ephemeris time (ET) and Universal Time Coordinated (UTC, which is also frequently called Spacecraft Event

Time, or SCET). This leapseconds kernel file is needed because time tags within an SPK file are in ET whereas SPACIT output uses the UTC times with which we are most familiar. The second utility file contains information about spacecraft clock partitions (resets) needed for conversions between normal spacecraft clock time (SCLK) and the encoded spacecraft clock time tags used in CK files.

The following dialogue illustrates the use of SPACIT to provide, in order, all of the summaries listed above.

```
------------------------------------------------------------------------

                        Welcome to SPACIT

          S, P, and C-kernel Information and Conversion Tool

------------------------------------------------------------------------

Select a task. (Enter the appropriate number, followed by <Return>.)

 1) Quit.
 2) Convert a text SPK or CK file to binary.
 3) Convert a binary SPK or CK file to text.
 4) Summarize all or part of a binary SPK file.
 5) Summarize all or part of a binary CK file.

5

Enter the name of the file to be summarized:

SPICEDISK:[CKFILES]VGR2NEP.CK

Enter name of the leapseconds kernel file (for time conversions):

SPICEDISK:[MISCKERNELS.LEAPSECONDS]LEAPSECONDS.KER

Enter name of the kernel file containing s/c clock partition information:

SPICEDISK:[MISCKERNELS.SCLK]SCLK.KER

Choose one. (Enter the number.)

 1) Return to previous menu.
 2) Summarize entire file.
 3) Summarize segments for a particular instrument.
 4) Summarize segments for a given time or time span (UTC).
 5) Summarize segments for a given time or time span (SCLK).

2
```

```
------------------------------------------------------------------------
Segment identifier: RET test 9/28/89
Instrument  : 32001
From SCLK   : 4   8966:31:000
To SCLK     : 4  12294:54:000
From UTC    : 1989 JUN 05 08:54:48.897
To UTC      : 1989 SEP 24 07:37:29.949
Reference   : J2000                          CK Type     :1
------------------------------------------------------------------------
------------------------------------------------------------------------
Segment identifier: RET test 9/28/89
Instrument  : 32002
From SCLK   : 4   8644:45:000
To SCLK     : 4  12295:09:000
From UTC    : 1989 MAY 25 15:30:21.774
To UTC      : 1989 SEP 24 07:49:29.949
Reference   : J2000                          Type      :1
------------------------------------------------------------------------

Choose one. (Enter the number.)

 1) Return to previous menu.
 2) Summarize entire file.
 3) Summarize segments for a particular instrument.
 4) Summarize segments for a given time or time span (UTC).
 5) Summarize segments for a given time or time span (SCLK).

3


Enter the NAIF integer spacecraft instrument ID:
(For example, 32000 for the Voyager 2 spacecraft bus,
              31001 for the Voyager 1 narrow angle camera, etc.)

32001


------------------------------------------------------------------------
Segment identifier: RET test 9/28/89
Instrument  : 32001
From SCLK   : 4   8966:31:000
To SCLK     : 4  12294:54:000
From UTC    : 1989 JUN 05 08:54:48.897
To UTC      : 1989 SEP 24 07:37:29.949
Reference   : J2000                          Type      :1
------------------------------------------------------------------------

Choose one. (Enter the number.)

 1) Return to previous menu.
 2) Summarize entire file.
 3) Summarize segments for a particular instrument.
 4) Summarize segments for a given time or time span (UTC).
 5) Summarize segments for a given time or time span (SCLK).

4


You may enter a single time, or two times representing an interval.
If you choose an interval, separate the two strings with an asterisk ('*').

(Some examples:  1999 AUG 14 21:10:55.067
```

```
                1999 AUG 14 * 1999 AUG 15
                1986-247//12:00:01.184
                29 February 1975 3:00
                Jan 1988 * Jan 1989          )

Enter all times in UTC:
```

*1989 JUL 04 * 1989 AUG 03*

```
------------------------------------------------------------------------
Segment identifier: RET test 9/28/89
Instrument   : 32001
From SCLK    : 4   8966:31:000
To SCLK      : 4  12294:54:000
From UTC     : 1989 JUN 05 08:54:48.897
To UTC       : 1989 SEP 24 07:37:29.949
Reference    : J2000                        Type      :1
------------------------------------------------------------------------
------------------------------------------------------------------------
Segment identifier: RET test 9/28/89
Instrument   : 32002
From SCLK    : 4   8644:45:000
To SCLK      : 4  12295:09:000
From UTC     : 1989 MAY 25 15:30:21.774
To UTC       : 1989 SEP 24 07:49:29.949
Reference    : J2000                        Type      :1
------------------------------------------------------------------------

Choose one. (Enter the number.)

 1) Return to previous menu.
 2) Summarize entire file.
 3) Summarize segments for a particular instrument.
 4) Summarize segments for a given time or time span (UTC).
 5) Summarize segments for a given time or time span (SCLK).
```

5

```
Enter the NAIF integer ID for this spacecraft (needed for clock encoding).
(Examples: -31 for Voyager 1,
           -32 for Voyager 2,
           -18 for Magellan, etc. )
```

-32

```
You may enter a single time, or two times representing an interval.
If you choose an interval, separate the two strings with an asterisk ('*').
Don't forget a partition number for each clock string.

(Some examples:  3 52045.54
                 4 12832:32:768 * 4 12832:59 )
```

*4 9021 * 4 12200:37*

```
------------------------------------------------------------------------
Segment identifier: RET test 9/28/89
Instrument   : 32001
```

```
From SCLK    : 4   8966:31:000
To SCLK      : 4  12294:54:000
From UTC     : 1989 JUN 05 08:54:48.897
To UTC       : 1989 SEP 24 07:37:29.949
Reference    : J2000                          Type      :1
-------------------------------------------------------------------------
-------------------------------------------------------------------------
Segment identifier: RET test 9/28/89
Instrument   : 32002
From SCLK    : 4   8644:45:000
To SCLK      : 4  12295:09:000
From UTC     : 1989 MAY 25 15:30:21.774
To UTC       : 1989 SEP 24 07:49:29.949
Reference    : J2000                          Type      :1
-------------------------------------------------------------------------

Choose one. (Enter the number.)

 1) Return to previous menu.
 2) Summarize entire file.
 3) Summarize segments for a particular instrument.
 4) Summarize segments for a given time or time span (UTC).
 5) Summarize segments for a given time or time span (SCLK).

1


Select a task. (Enter the appropriate number, followed by <Return>.)

 1) Quit.
 2) Convert a text SPK or CK file to binary.
 3) Convert a binary SPK or CK file to text.
 4) Summarize all or part of a binary SPK file.
 5) Summarize all or part of a binary CK file.

1
```

Integrating SPACIT functionality into your own program

SPACIT is built around a few high-level SPICELIB subroutines which do the file conversion and file summary work. Those SPICE system users who wish to integrate the SPACIT utility functions into their own, custom program should find it easy to use the SPACIT main program source code as a guide for doing so.