

DTIC COPY

Plasma Interactions with Spacecraft (II)

V. A. Davis
M. J. Mandell

Science Applications International Corporation
10260 Campus Point Drive
San Diego, CA 92121

Scientific Report No. 2

1 Apr 2009

Approved for public release; distribution unlimited.



AIR FORCE RESEARCH LABORATORY
Space Vehicles Directorate
29 Randolph Road
AIR FORCE MATERIEL COMMAND
HANSCOM AFB, MA 01731-3010

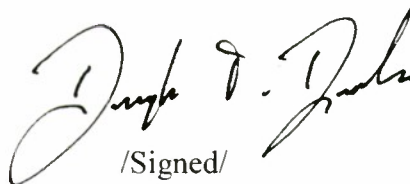
This technical report has been reviewed and is approved for publication.

AFRL-VS-HA-TR-2009-1050 (II)



/Signed/

ADRIAN WHEELOCK
Contract Manager



/Signed/

DWIGHT T. DECKER, Chief
Space Weather Center of Excellence

This report has been reviewed by the ESC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS).

Qualified requestors may obtain additional copies from the Defense Technical Information Center (DTIC). All others should apply to the National Technical Information Service.

If your address has changed, if you wish to be removed from the mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/VSIM, 29 Randolph Rd., Hanscom AFB, MA 01731-3010. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is published in the interest of scientific and technical information exchange and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.					
1. REPORT DATE (MM-DD-YYYY) 04-01-2009		2. REPORT TYPE Scientific Report No. 2		3. DATES COVERED (From - To) 03-01-2007 to 03-31-2009	
4. TITLE AND SUBTITLE Plasma Interactions with Spacecraft N2kDB Database and Memory Management Software for Nascap-2k, Version 1.0 Draft Documentation				5a. CONTRACT NUMBER FA8718-05-C-0001	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER 63401F	
6. AUTHOR(S) V.A. Davis, N.R. Baker, B.G. Gardner, R.A. Kuharski, M.J. Mandell, A.J. Ward, K.G. Wilcox				5d. PROJECT NUMBER 5021	
				5e. TASK NUMBER RS	
				5f. WORK UNIT NUMBER A1	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Science Applications International Corporation 10260 Campus Point Drive, Mailstop A-2A San Diego, CA 92121				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory 29 Randolph Road Hanscom AFB, MA 01731				10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RVBXR	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S) AFRL-RV-HA-TR-2009-1050 (II)	
12. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited.					
13. SUPPLEMENTARY NOTES					
20100315115					
14. ABSTRACT Nascap-2k is a spacecraft charging and plasma interactions code designed to be used by spacecraft designers, aerospace and materials engineers, and space plasma environments experts to study the effects of both the natural and spacecraft-generated plasma environment on spacecraft systems. N2kDB is the new database and memory management system developed for Nascap-2k.					
15. SUBJECT TERMS Nascap-2k, Potentials, Space environment, Spacecraft, Spacecraft charging, DSX, N2kDB					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT SAR	18. NUMBER OF PAGES 111	19a. NAME OF RESPONSIBLE PERSON Adrian Wheelock
a. REPORT UNCLASSIFIED	b. ABSTRACT UNCLASSIFIED	c. THIS PAGE UNCLASSIFIED			19b. TELEPHONE NUMBER (include area code) (781) 377-9668

CONTENTS

	Page
1 Introduction.....	1
1.1 Intended Audience	1
1.2 Document Summary	1
2 Requirements Document.....	1
3 Design Considerations	2
3.1 SQLite versus SAIC-Developed Database Software Core	2
3.2 Simple Memory Manager in C++ versus Fortran 90 Dynamic Memory Allocation.....	3
3.3 Additional Requirements	3
4 Architecture.....	4
4.1 Global Structure.....	4
4.1.1 Interim Global Structure	5
4.2 Database File Structure.....	6
4.2.1 File Type	6
4.2.2 Data Item Names.....	6
4.2.3 File Contents	7
4.3 Database Access.....	7
4.3.1 Data Initialization.....	7
4.3.2 String Comparisons.....	7
4.3.3 Functional Requirements	7
4.3.4 N2kDB Functions	7
4.3.5 N2kDB Internal Structure.....	8
4.3.6 N2kDB Tool	8
5 Technical Issues	8
5.1 Operating System.....	8
5.2 Source Code Control.....	8
5.3 Coding Standards	9
5.3.1 C++	9
5.3.1.1 Style	9
5.3.1.2 Functionality	10
6 Detailed Design.....	10

6.1	Database Access.....	10
6.1.1	Open Database	11
6.1.2	Close Database.....	12
6.1.3	Memory Management.....	12
6.1.4	Grid Functions	12
6.1.5	MSIO File Index	12
6.1.6	Material, Species, and Grid Functions.....	13
6.1.7	History Function	13
6.1.8	Error Handling	13
6.1.9	Sizes	13
6.1.10	Other	14
6.2	N2kDB Internal Structure.....	14
6.2.1	Error Handling	18
6.2.2	MemoryManager Class.....	18
6.2.3	N2kDBDataModel Class	18
6.2.3.1	Open Database	19
6.2.3.2	Close Database.....	19
6.2.3.3	Read and Write	19
6.2.3.4	Other	19
6.2.3.5	Material, Species, and Grid functions.....	19
6.2.3.6	Grid Functions	20
6.2.3.7	Sizes	20
6.2.4	History Function	21
6.2.4.1	Data Items	21
6.2.4.2	Errors.....	24
6.2.5	Database Class	25
6.2.6	GridData Class	28
6.2.7	MSIO Class.....	31
6.2.7.1	MSIO Files.....	37
6.3	N2kDB Tool	38
6.3.1	User Interface.....	39
6.3.2	Code Structure	42
6.3.2.1	N2kDB Tool library.....	42

6.3.2.2	N2kDB Tool Java interface	44
6.3.2.3	N2kDB Tool Console User Interface	48
7	Testing	48
7.1	Testing of Each Class	48
7.1.1	Unit Test Code Guidelines	48
7.1.2	MSIO Unit Tests	49
7.1.2.1	Test 4.....	49
7.1.2.2	Test 5.....	49
7.1.2.3	Test 6.....	50
7.1.2.4	Test 7.....	50
7.1.2.5	Test 11.....	50
7.1.2.6	Test 12.....	50
7.1.2.7	Test 13.....	50
7.1.2.8	Test 15.....	50
7.1.2.9	Test 15b.....	51
7.1.2.10	Test 16.....	51
7.1.2.11	Test 17.....	51
7.1.2.12	Test 18.....	51
7.1.2.13	Test 19.....	51
7.1.2.14	Test 23.....	51
7.1.2.15	Test 32.....	52
7.1.3	N2kDBDataModel Tests	52
7.1.3.1	Test #1.....	52
7.1.3.2	Test #2.....	52
7.1.3.3	Test #3.....	52
7.1.3.4	Test #4.....	52
7.1.3.5	Test #5.....	52
7.1.3.6	Test #6.....	52
7.1.3.7	Test #7.....	52
7.1.3.8	Test #8.....	53
7.1.3.9	Test #9.....	53
7.1.3.10	Test #10.....	53
7.1.3.11	Test #11.....	53

7.1.3.12 Test #12.....	53
7.1.3.13 Test #13.....	53
7.1.3.14 Test #14.....	53
7.1.3.15 Test #15.....	53
7.1.3.16 Test #16.....	53
7.1.3.17 Test #17.....	53
7.1.3.18 Test #18.....	53
7.1.3.19 Test #19.....	54
7.1.3.20 Test #20.....	54
7.1.3.21 Test #21.....	54
7.1.3.22 Test #22.....	54
7.1.3.23 Test #23.....	54
7.1.3.24 Test #24.....	54
7.1.3.25 Test #25.....	54
7.1.3.26 Test #26.....	54
7.2 Consistency with Specification.....	54
8 Intermediate Implementation of N2kDB	55
References.....	57
Appendix A. <i>Nascap-2k 3.2</i> Structure.....	59
A.1 DBLib	60
A.1.1 Buffio	60
A.1.2 Dbdata.....	60
A.1.3 Dbinfo	61
A.1.4 Dbfile	61
A.1.5 DbLib Errors	61
A.1.6 DbLib Commands.....	63
A.1.7 Summary of DBLib Functionality Used.....	66
A.2 DynaBase	67
A.3 Data Requests from Nascap-2k Java Interface	67
A.4 Data.....	68
A.4.1 Summary Comments.....	73
A.5 Data Storage.....	73
A.6 Searches Contemplated.....	74

Appendix B. New Data Item Names.....	75
Appendix C. N2kDbTest	79
C.1 Details of N2kDBTest	79
Appendix D. <i>Nascap-2k</i> Testing.....	83
Appendix E. Requirements Verification.....	85
E.1 Introduction.....	85
E.1.1 Purpose.....	85
E.1.2 Project Scope	85
E.2 Overall Description.....	85
E.2.1 Product Perspective.....	85
E.2.2 Product Features.....	85
E.2.3 Operating Environment.....	85
E.2.4 Design and Implementation Constraints.....	86
E.2.5 User Documentation	86
E.3 System Features	86
E.3.1 Data Storage Capacity and Format	86
E.3.1.1 Description and Priority.....	86
E.3.1.2 Functional Requirements	86
E.3.1.2.1 Maximum Database File Size.....	86
E.3.1.2.2 Maximum Record Size	86
E.3.1.2.3 Maximum Rows in a Table.....	86
E.3.1.2.4 Data Format (ASCII, XML, binary,...)	87
E.3.1.2.5 Data in Single or Multiple Files.....	87
E.3.1.2.6 File Sharing.....	87
E.3.1.2.7 Need for Standard Access Format	87
E.3.1.2.8 Data Structure	87
E.3.1.2.9 Error handling requirements	87
E.3.2 Data Transfer Rate	87
E.3.2.1 Description and Priority.....	87
E.3.2.2 Functional Requirements	88
E.3.2.2.1 Size and Frequency of Reads and Writes	88
E.3.2.2.2 Allowable Impact on Calculation Time.....	88
E.3.2.2.3 Error Handling Requirements.....	88
E.3.3 Memory Management.....	88
E.3.3.1 Description and Priority.....	88
E.3.3.2 Functional Requirements	88

E.3.3.2.1	Maximum Memory Required	88
E.3.3.2.2	Allowable Impact on Calculation Time.....	88
E.3.3.2.3	Error Handling Requirements	88
E.3.3.2.4	Data Size Determination	88
E.3.3.2.5	Multiprocessor Operation	89
E.3.4	Data Access.....	89
E.3.4.1	Description and Priority	89
E.3.4.2	Functional Requirements	89
E.3.5	Support of Pre-Existing Databases	89
E.3.6	Data Structure	89
E.4	External Interface Requirements.....	90
E.4.1	User Interfaces	90
E.4.2	Hardware Interfaces	90
E.4.3	Software Interfaces	90
E.4.4	Communications Interfaces	90
E.5	Additional Requirements from Section 3.3 of this Document.....	90
E.6	Appendix References	90
Appendix F	Implementation Plan	93
F.1	Schedule	93
F.2	Tasks	93
F.3	Revisions Needed to FORTRAN Portion of <i>Nascap-2k</i>	94
Appendix G	Open Issues	97

FIGURES

	Page
Figure 1. Anticipated structure of <i>Nascap-2k 4.1</i> main components.....	4
Figure 2. Intermediate implementation code structure.	5
Figure 3. Components of N2kDB in relation to the database file, <i>Nascap-2k</i> , and the supporting tools N2kDB Tool and N2kDBTest	15
Figure 4. Structure of MSIO class.	34
Figure 5. Screen that appears when an MSIO file is opened.....	40
Figure 6. Screen that appears when a <i>Nascap-2k</i> database is opened	41
Figure 7. Screen that appears when button on Figure 5 or Figure 6 is clicked.....	42
Figure A1. <i>Nascap-2k 3.2</i> structure of Main components.....	59
Figure C1. N2kDBTest Java interface.	80

TABLES

	Page
Table 1. Comparison of benefits and drawbacks to using SQLite vs. MSIO as the database software core.	3
Table 2. N2kDB classes.	8
Table 3. N2kDB Application Programmer Interface.	11
Table 4. How to get number of items from N2kDB	14
Table 5. Classes and structs of N2kDB	17
Table 6. Public methods of the MemoryManager class.	18
Table 7. Public methods of N2kDBDataModel class.	21
Table 8. Selected private methods of N2kDBDataModel class.	23
Table 9. Error codes.	24
Table 10. Public Database class methods.	26
Table 11. Selected private Database class methods.	28
Table 12. Public methods of GridData class.	29
Table 13. Selected private member functions of GridData class.	31
Table 14. Public methods of MSIO class.	32
Table 15. Public member functions of MSIO:Header class.	36
Table 16. Content of MSIO metadata record.	38
Table 17. N2kDB Tool Library.	43
Table 18. Methods of N2kDB Tool class of N2kDB Tool Java interface.	45
Table 19. Native methods used by N2kDB Tool Java interface.	46
Table 20. Methods of Java DataPanel class of N2kDB Tool Java interface.	48
Table A1. Major components of <i>Nascap-2k 3.2</i>	60
Table A2. Error messages reported by DbLib subroutines.	62
Table A3. Error messages reported by existing MSIO	63

Table A4. Keywords used in present database commands.	64
Table A5. Information returned from <code>dbinfo(Inquire Grid)</code> commands.	66
Table A6. DynaBase entry points.	67
Table A7. Data requests from user interface.	68
Table A8. Data items associated with surfaces.	69
Table A9. Data items associated with surface nodes.	69
Table A10. Data items associated with volume elements.	70
Table A11. Data items not associated with surfaces or volume elements.	71
Table A12. Quantities that are used as indexes.	73
Table B1. Data items associated with surfaces.	76
Table B2. Data items associated with surface nodes.	76
Table B3. Data items associated with volume elements.	77
Table B4. Data items not associated with surfaces or volume elements.	78
Table C1. Contents of <code>N2kDBTest.DP</code> after execution.	81
Table C2. Contents of <code>N2kDBTest.PT1</code> file.	82
Table C3. Differences between <code>N2kDBTest.PT1</code> and <code>N2kDBTest.PT2</code>	82
Table F1. Implementation steps.	93
Table F2. Tasks to be completed.	94

1 INTRODUCTION

Nascap-2k is a spacecraft charging and plasma interactions code designed to be used by spacecraft designers, aerospace and materials engineers, and space plasma environments experts to study the effects of both the natural and spacecraft-generated plasma environment on spacecraft systems. **N2kDB** is the new database and memory management system developed for *Nascap-2k*.

Background material can be found in *Software Requirements Specification for Nascap-2k Database and Memory Manager* and in appendices to this document.

1.1 Intended Audience

This document is intended primarily for use by software development teams of products that use N2kDB.

1.2 Document Summary

This document describes the design of **N2kDB**, the new *Nascap-2k Database and Memory Management Software*.

Section 3 describes the general design considerations. The architecture of the code is described in Section 4. Some additional technical issues are discussed in Section 5, and Section 6 describes the detailed design.

The testing program is described in Section 7 and the intermediate implementation is described in Section 8.

Appendix A describes the database used in earlier versions of *Nascap-2k*¹.

Appendix B lists the new data item names. Appendix C describes the test program developed to verify that **N2kDB** has the desired functionality, and Appendix D describes the procedure for verifying correct behavior in *Nascap-2k*.

Appendix E is an itemized list of how **N2kDB** meets the requirements.

The implementation plan is described in Appendix F, and open issues are listed in Appendix G.

2 REQUIREMENTS DOCUMENT

This document builds on the *Software Requirements Specification for Nascap-2k Database and Memory Manager*, which appears as an appendix to the first interim report for this contract, AFRL-VS-HA-TR-2007-1062.²

3 DESIGN CONSIDERATIONS

Software Requirements Specification for Nascap-2k Database and Memory Manager provides background material and describes most of the considerations important in the design of the new software. Some general design considerations, assumptions, dependencies, constraints, and goals are discussed below.

The database and memory management tasks are performed separately, and both function on all *Nascap-2k*-supported platforms. The database is accessible from FORTRAN with Cray style pointers, C++, and Java. The database supports all database call capabilities used by *Nascap-2k* 3.2 (the last version to be released with the old database) and desired for *Nascap-2k* 4.1 (the first version to be released with the new database.).

3.1 SQLite versus SAIC-Developed Database Software Core

We considered two approaches to database software. We considered the public domain code package **SQLite** and our own **MSIO**.

SQLite is a public domain C software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. Bindings (separate code packages that talk to the **SQLite** C code from another language) exist for Fortran, C++, and Java. It appears to be possible to access an **SQLite** database from Microsoft EXCEL or ACCESS using ODBC. Our initial assessment is that **SQLite** is the only SQL access tool amenable for use in *Nascap-2k*. Others have licensing difficulties, allow only one process to access the database at once, or require client-server operations.

Nascap-2k 3.2 uses **MSIO** (Mass Storage Input and Output, a Fortran and C simulation of CDC Fortran commands developed by SCUBED in the 1970s) for reading from and writing to the random access database file. The set of C functions, *fastio*, performs the opening, closing, reading, and writing. A set of four Fortran subroutines, *OPENMS*, *CLOSMS*, *READMS*, and *WRITMS* maintains and uses the user-allocated index that keeps track of the size and location of each piece of data in the database. Each piece of data is identified either by a unique string (name key file) or by the location of its specification in the index (number key file).

Before making a decision, we tested the use of **SQLite** through a prototype implementation. While speed was found to be an issue, preliminary testing shows that the use of BLOBs (Binary Large Object) for large data structures such as the grid definitions improves speed significantly. It is known that, with the new database, a large fraction of the database calls will not be needed and minimizing those is expected to improve speed still more. The requirement on speed for the new database is that it uses no more than a comparable fraction of total computation time (~5%) as the current database implementation uses.

With either **SQLite** or **MSIO** we would access the database from the Java code by calling a C++ dynamically linked library through Java Native Interface (JNI).^{3, 4, 5, 6}

After considering the issues noted in Table 1 below, we decided to rewrite **MSIO** to overcome its present limitations and use it for the new *Nascap-2k* database.

Table 1. Comparison of benefits and drawbacks to using SQLite vs. MSIO as the database software core.

	PROS	CONS
SQLite	Truly open-source with no restrictions. Standard database access approaches. Standard access from multiple languages. Access using commercial SQLite database access codes possible.	Slow (can be mitigated by using BLOBs and minimizing number of database calls). Code comprehensibility presently unknown. General code rather than tailored. SQLite bindings may not be strictly cross platform.
MSIO	Source code is our own. Speed. Fastest possible database access. Optimized for <i>Nascap-2k</i> . Intimate knowledge of code going forward. Transparency to other developers at Hanscom. Ensures that code is cross platform.	Does not conform to any commonly recognized standard. May be necessary to create capabilities that are already available in SQLite . All new code: no testing history.

3.2 Simple Memory Manager in C++ versus Fortran 90 Dynamic Memory Allocation

A simple C++ memory manager was written during the proof-of-concept prototyping. It was decided to expand on this approach. If speed requires, the approach can be modified to allow for more explicit control.

3.3 Additional Requirements

In addition to those requirements given in *Software Requirements Specification for Nascap-2k Database and Memory Manager*, we have the following requirements:

- *Nascap-2k* also operates on an Apple Macintosh Pro running OS X Unix (10.5 (Leopard)) with 2 dual core 64-bit Intel Xeon CPUs for a total of 4 processors. In this environment, Intel compilers are used.
- *Nascap-2k* is, so far as we know, compatible with Windows Vista and Windows 7, and has been superficially tested.
- All of the new software will be written in a cross platform manner as much as possible.
- The *Nascap-2k* database file will be platform independent.
- The software structure and internal and external documentation will be such that the software is easy to maintain. Specifically, all code and documentation will be kept together in the SourceSafe database, coding standards will be developed and employed, appropriate levels of encapsulation and generality will be used to simplify analysis of the science code operations,

code structures and data storage structures in which variables cannot be viewed in a debugger will be avoided, thorough documentation will be written in concert with code development, and all documentation will be understandable to programmers unfamiliar with database jargon.

4 ARCHITECTURE

4.1 Global Structure

The *Nascap-2k* 4.1 structure will be as shown in Figure 1. Database access will be accomplished through the new dynamically linked library **N2kDB**. (A dynamically linked library with JNI is needed for access from the Java user interface) All of the computational modules, along with the user interface, will communicate directly with **N2kDB**. **N2kDB** is also accessible independently through **N2kDB Tool**.

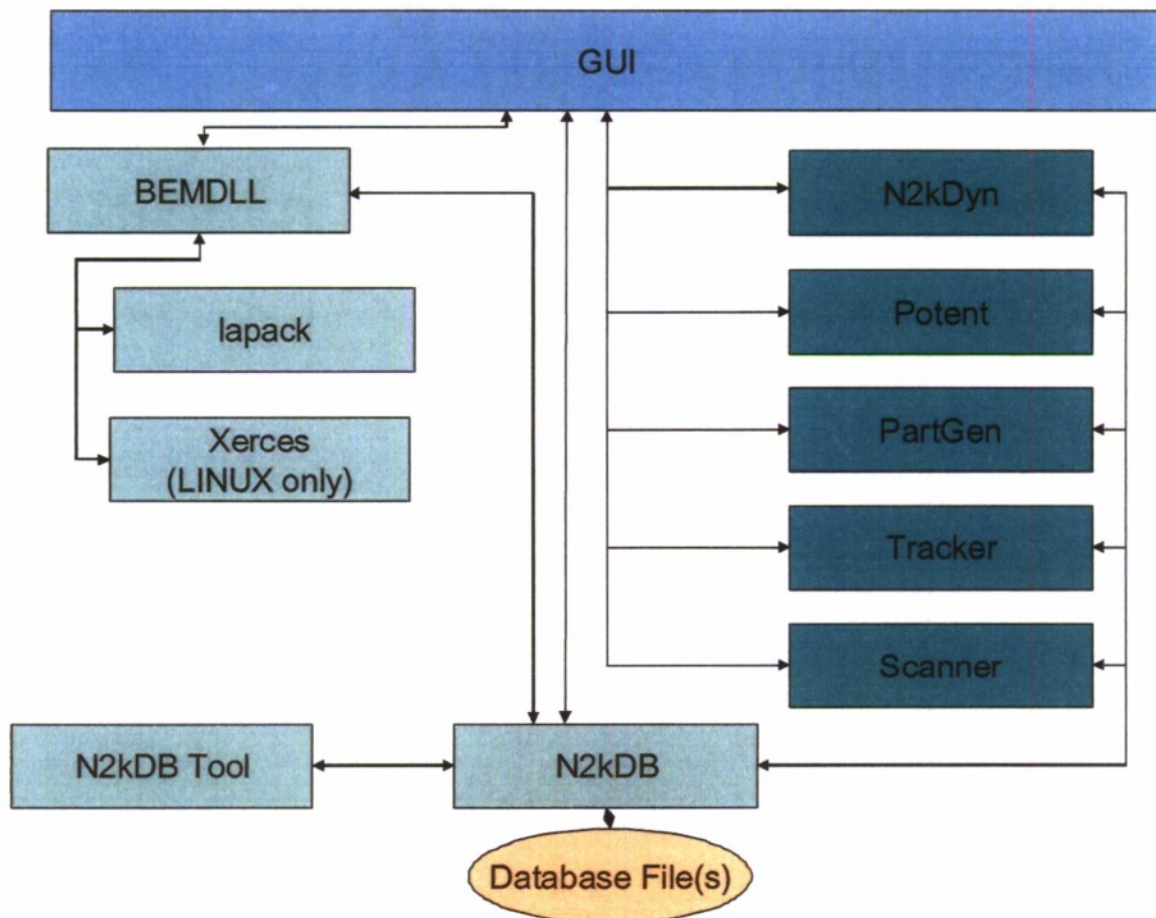


Figure 1. Anticipated structure of *Nascap-2k* 4.1 main components.

With the new structure, the separate **DynaBase** dynamically linked library (which was mainly a wrapper from the C++ science coding to the Fortran database coding) will be unnecessary. In addition to handling reading from and writing to the database, it restructures some of the data to fit the data organization within **BEMDLL** or the user interface. This functionality will be accomplished by C++ and Java classes within **BEMDLL** and the user interface.

4.1.1 Interim Global Structure

As an interim step, the code structure shown in Figure 2 has been implemented. This version of *Nascap-2k* is 4.0.#, where # indicates successive intermediate versions as the new database is more fully implemented. Fortran subroutines were written (dbdata, dbfile, dbinfo, and buffio) that take the keyword list arguments used by *Nascap-2k* 3.2. and return pointers to the same locations within **N2kDB**.

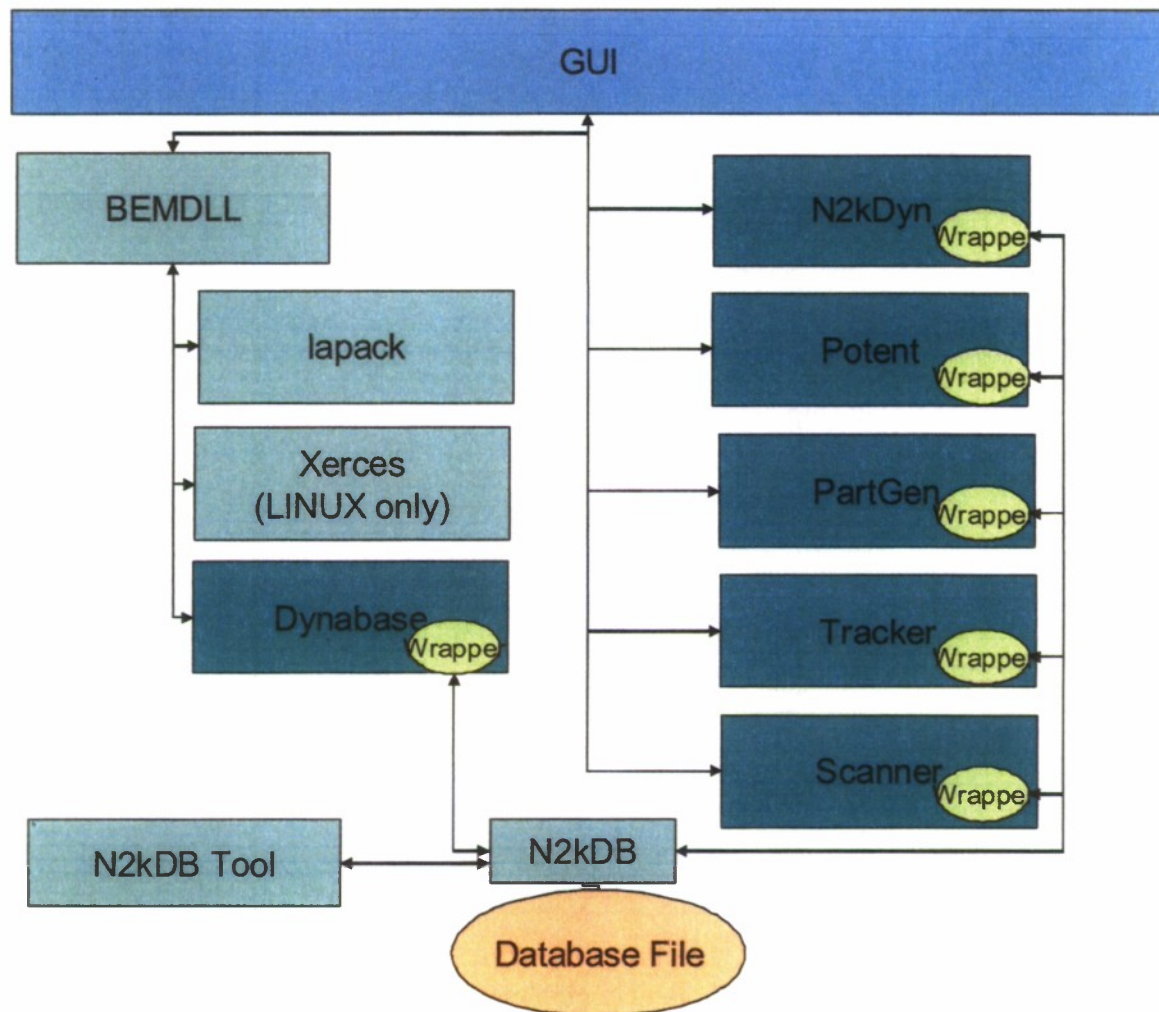


Figure 2. Interim implementation code structure.

4.2 Database File Structure

4.2.1 File Type

N2kDB uses updated **MSIO** random access files. MSIO has been rewritten using modern C++ object-oriented coding techniques. We use 4-word keys for each record, along with the words specifying the record's length and location. We also added a new type of key to MSIO, a hybrid key. A hybrid key is a combination of a name key and number key in the record immediately following.

Metadata, such as size of the index, size of a sector, type of index, database version number, etc., is stored before the beginning of the index.

4.2.2 Data Item Names

Each item stored has a unique case-insensitive 16 character (4 words) identifier except for, time dependent data, which is stored in a file that uses 16 characters and an integer in the following word as an identifier (5 words). The choice of 16 characters balances the clarity that can be achieved with longer data item names with the desire for compactness and the nuisance of extra blanks. The integer gives the time index. In the file index, the characters are strictly upper case in order to avoid the complications of mixed-case. The characters are converted to upper case on reading and writing, so that the rest of the code can be insensitive to case. The database software knows which data item is stored in which file and uses the index of the appropriate file to locate the data.

Each gridded data item has a data item name of up to 12 characters. The data item name for database access is the 12 characters, an underscore, and three digits that specify the grid. (12characters_####) Gridded data has keywords that start with "GEIm," "GNod," or "Gnod8," for element centered data, 4 value per node data, and 1 value per node data.

Data items names for data associated with surfaces start with "Srf_". Data items names for data associated with nodes start with "Nod_". Data item names for data associated with conductors start with "Cnd_".

Each material, grid, and species will be stored in a single record. (In the interim implementation, the materials and species records are not initially implemented.) The data item names are Material_####, Grid_####, and Species_#. This convention limits the number of materials and the number of grids to 999 and the number of species to 99. The first item in each material and species record is the name. Allowance is made for names up to 80 characters.

For special element information, the data item names are MtrxElems_##### and BndSrfcs_#####, where ##### is the special element number. This convention limits the number of special elements to 99999.

The particle data will be saved with data item names of the form Prtcls_S##_****, where ## is the species number and **** is the page number. This convention limits the number of pages to 9999. (In the interim implementation, the particle data storage has not been revised.) The only information presently stored in the particle file header that is not redundant is IPType. IPType

will be added to the /ActPrt/ common block. This will allow different type particles of the same species to exist.

Data item names for miscellaneous data start with “Gen_”. The data item name for the general problem data is “General”.

4.2.3 File Contents

The database consists of up to four types of files: *prefix.NDB*, *prefix.NSE*, *prefix.NPT##*, and *prefix.NTM*. These files contain general information, special elements, pages of particles, and time dependent information respectively. This approach allows the user to share special element information between projects, to arbitrarily delete all the particles of a given species, and to arbitrarily delete time history information.

Data with data item names of the form *Prtcls_S##_***** are stored in the file *prefix.NPT##*. Data with data item names of the form *MtrxElems_#####* and *BndSrfcs_#####* are stored in the file *prefix.NSE* (special elements), Time dependent data is stored in the file *prefix.NTM*. All other data is stored in *prefix.NDB*.

The NDB file allows for 10,000 keys. The NSE file allows for 100,000 keys. The NPT file allows for 100,000 keys. The NTM file allows for 100,000 pairs of hybrid keys.

4.3 Database Access

4.3.1 Data Initialization

Initialization of values is *not* to be handled by the database software. The existing subroutines *s3zero* and *s3set* are used for initialization.

4.3.2 String Comparisons

All string comparisons are case insensitive.

4.3.3 Functional Requirements

Nascap-2k has functional requirements of file access that have been addressed in the design of database access code. These are enumerated in Section 3.4.2 of *Software Requirements Specification for Nascap-2k Database and Memory Manager*.

4.3.4 N2kDB Functions

The **N2kDB** functions that are called directly by Fortran, by Fortran wrapper functions, by a C++ class within **BEMDLL**, and by a Java class in the interface are tabulated in Section 6.1. All functions for which access from Java is contemplated have JNI wrappers. All functions have wrappers that convert Fortran character variables to null-terminated strings. All functions return an integer that specifies any errors. Memory for grid array data is allocated by **N2kDB**. Memory for all other data is handled by the calling routine. The **MSIO** index is written by each write function.

4.3.5 N2kDB Internal Structure

N2kDB is written in C++.

N2kDB has an outer shell with the methods accessible from Fortran, C++, and Java. The wrapper subroutines of N2kDB call members of the N2kDBDataModel and MemoryManager classes. The list of classes appears in Table 2.

Table 2. N2kDB classes.

Class name
N2kDB
N2kDBDataModel
Memory Manager
Database
GridData
MSIO

4.3.6 N2kDB Tool

N2kDB Tool reads and writes individual data items, examines the index and pulls out tables of numbers.

N2kDB Tool provides a user interface directly to a *Nascap-2k* database. It uses N2kDB. It allows the user to view (read only) the database structure, the MSIO index—including ASCII data item names and record location and sizes. It allows the user to see any metadata that describes the database. The code allows read/write access to individual records.

5 TECHNICAL ISSUES

5.1 Operating System

N2kDB was being developed with the same operating system and compilers as *Nascap-2k*. It is fully compatible with Win32, Red Hat LINUX and multiprocessor MacOS X UNIX (10.5) environments. The MacOS X environment is used to test multiprocessor and 64-bit operations.

5.2 Source Code Control

All of the new software is kept in Source Safe with *Nascap-2k*. As with *Nascap-2k*, makefiles for LINUX use are also kept under Source Safe control.

5.3 Coding Standards

To the greatest extent possible, all new coding is in C++, with externally exposed method calls. The exposed methods will consist of language appropriate wrappers accessible from Fortran, C++, and Java. Java access is provided via JNI wrappers. Fortran coding is permissible only when necessary to integrate with existing legacy Fortran-77 code. Fortran-90 may be used provided it is structured to integrate smoothly with existing legacy Fortran-77 code.

5.3.1 C++

To the extent practical, a uniform style and standards were used. In addition to those of the SAIC Monitoring Systems Division, the following standards were followed.

5.3.1.1 Style

All instance variables will be private. Getters and setters will provide access to them. Methods, properties, and const fields can be public or private, as appropriate.

Every method will be preceded by a documentation comment explaining the purpose of the method. Every public property will be preceded by a documentation comment explaining the purpose of the property and describing the value it represents.

Parameter names will be explicit, especially if they are integers or Booleans.

Each variable will be defined just before it is used for the first time rather than at the beginning of the block.

Standard structural constructs will be used as follows:

Use the `if` statement to select a statement for execution based on the value of a Boolean expression.

Use the `for` statement only when a variable starts from a beginning value, and is incremented or decremented by a constant.

Use the `foreach` statement to iterate through a collection that does not need to be changed.

Use the `while` statement to execute a statement until a specified expression evaluates to `false`.

Use the `do` statement to execute a statement repeatedly until a specified expression evaluates to `false`, at least one time even if the expression never evaluates to `true`.

Use the `switch` statement to handle multiple selections by passing control to one of the `case` statements within its body.

Use the `try-catch` statement to handle raised exceptions. All code that can throw an exception must be wrapped in a `try-catch` statement:

Use Pascal casing (The first character is upper case, and the first letter of each word is also upper case) for classes and Camel casing (The first character is not upper case, but the first letter of each word is upper case) for methods, properties, and variables.

Use the Visual Studio defaults for tab stops and code formatting. Use blank lines freely to separate parts of a method that are logically distinct. Opening curly braces will always be placed on the next line after the keyword with which they are associated. Keywords and closing curly braces must always line up vertically.

5.3.1.2 Functionality

Appropriate levels of encapsulation and generality will be used to simplify analysis of the science code operations. Code structures and data storage structures in which variables cannot be viewed in a debugger will be avoided.

Additionally, the C++ standard library will be used as much as possible. For example, the C++ string class will be used rather than char arrays, and the standard library's data structure classes (such as vector, list, or map) will be used rather than creating new classes.

6 DETAILED DESIGN

This section describes the design of **N2kDB** and **N2kDB Tool**.

6.1 Database Access

N2kDB is a dynamic linked library with functions that are called directly by Fortran, by Fortran wrapper functions, by a C++ class within **BEMDLL**, or by a Java class in the interface. These functions are given in Table 3. The argument "void *data" is the memory location at which the data is located. These functions have the following characteristics:

- All functions for which access from Java is contemplated have JNI wrappers. Among numerous other issues, the JNI wrappers convert single precision values stored in the database to double precision values.
- All functions with char* arguments that require Fortran access have Fortran callable wrappers with room for the length of the character variable in the argument list, as passed by Fortran. The wrapper then converts the character variable to a null-terminated char* for use by the C++ coding.
- All functions return an integer that specifies an error code, if any.
- When "nwords" is included in the argument list of a Read function, the value before the call is the maximum value and the return value is the actual number of words read.
- The calling routine is responsible for the allocation of space, except for arguments given as "void** data," in which case N2kDB allocates the necessary space.

Table 3. N2kDB Application Programmer Interface.

Function name and arguments
AllocateMemory(int* nwords, void** data)
AllocateMemoryGrid(int* dimension, int* grid, void** data)
ClearDeadSpace()
CloseDatabase()
DBErrorMessage(int* messageNumber, char** ErrorMessage)
DeleteAllGrids()
DeleteMaterial(char* name)
DeleteSpecies(char* name)
FreeMemory(void *data)
GetCount(char* name, int* number)
GetLength(char* name, int* number)
GetMaterials(int* nwords, vector char* names)
GetSpecies(int* nwords, vector char* names)
OpenDatabase(char* prefix, char* version)
ReadDatabase(char* name, int* nwords, int* time, void *data)
ReadDatabaseGrid(char* name, int*dimension, int* grid, int* time, void** data)
ReadGridData(int* number, int* nwords, void *data)
ReadHistory(char* name, int* entries, int* numberOfEntries, int* nwords, void *data)
ReadMaterialData(char* name, int* nwords, void *data)
ReadSpeciesData(char* name, real* charge, real* mass, real* fraction)
WriteDatabase(char* name, int* nwords, int* time, void *data)
WriteDatabaseGrid(char* name, int* dimension, int* grid, int* time, void *data)
WriteGridData(int* number, int* nwords, void *data)
WriteMaterialData(char* name, int* nwords, void *data)
WriteSpeciesData(char* name, real* charge, real* mass, real* fraction)

6.1.1 Open Database

OpenDatabase has the following functionality.

- It opens the database and initializes the memory manager.
- Only one database can be open at one time.
- It creates the database if it does not already exist.
- It specifies the files that form the database.

- It reads and verifies the index.
- It verifies compatibility of database and MSIO version numbers. The version number in the argument list should identify the code as well as have an incremental number.
- Optional read only access to the database is accommodated, so that **N2kDB Tool** can be used simultaneously. It appears that on some operating systems, it is possible for two processes to have the same file open for read/write access.

6.1.2 Close Database

`CloseDatabase` closes the database

`CloseDatabase` releases all allocated memory.

In general, the index is *not* written by `CloseDatabase`.

6.1.3 Memory Management

`AllocateMemoryGrid`, `AllocateMemory`, and `FreeMemory` allocate and release memory for gridded quantities, special element data, and perhaps for pages of particles without reading from or writing to the database. `AllocateMemoryGrid` is used on creation of a new quantity (rather than reading it from the database).

`ReadDatabaseGrid` also handles memory management. If the requested data does not exist in the database, `ReadDatabaseGrid` initializes the allocated space to zero.

The calling subroutine is responsible for allocation of space for all other data. `AllocateMemory` may be used.

The calling program is responsible for calling `FreeMemory` to release any allocated memory. Additionally, `CloseDatabase` releases all memory.

N2kDB does *not* track which data item is associated with which memory location.

6.1.4 Grid Functions

For `ReadDatabaseGrid`, `WriteDatabaseGrid`, and `AllocateMemoryGrid`, the dimension is the number of items per grid point. The code underneath keeps track of the grid size.

6.1.5 MSIO File Index

The index is written after any operation that modifies the index and not otherwise. When writing a record that will not fit in the existing space, the new record is written at the end of the file, leaving the old data as dead space.

`ClearDeadSpace` copies data within the database to remove all the dead space. We need to consider under what circumstances `ClearDeadSpace` should execute. It should *not* be done every time the database is closed.

6.1.6 Material, Species, and Grid Functions

The `DeleteAllGrids` function deletes the matrix elements (data item names `MtrxElems_#####` and `BndSrfcs_#####`) as well as the grid structure for all the grids.

The `Read` and `Write` `Grid`, `Material`, and `Species` data functions are separate functions for convenience. Almost the same functionality can be obtained from using the `ReadDatabase` and `WriteDatabase` functions for the appropriate data item name. These functions free *Nascap-2k* from dealing directly with the data storage structure. Grids are addressed by number and `Materials` and `Species` are addressed by name. The amount of information associated with each grid and each material is specified by the calling code.

`GetMaterials` and `GetSpecies` return arrays with contents that are the material and species names respectively.

6.1.7 History Function

`ReadHistory` returns time histories of potentials and the various currents for a specified list of surfaces. It is only called from Java and C++ and it allocates space for the data returned.

The surface number is the Fortran number. The first one is “1”.

6.1.8 Error Handling

The first argument of `DBErrorMessage` is the input--and is the return value of a `N2kDB` function. The `char**` argument is the return value, which is the error message to which the first argument corresponds. As the return value is a `char**` for which space is specified in the wrapper, the C++ and Java calling codes do not need to allocate space. The Fortran wrapper truncates a message that is too long.

6.1.9 Sizes

The `GetLength` and `GetCount` functions are used to get the number of various items. They use the index to obtain the number of surfaces, nodes, conductors, special elements, species, materials, grid, timesteps, and pages of particles. Table 4 specifies how these functions are used to get useful quantities. The `GetCount` function gets the number of items with data item names that start with the specified string.

Table 4. How to get number of items from N2kDB.

Function	Specifics
Number of Surfaces	Length of 'Srf_Element' record/4
Number of Nodes	Length of 'Nod_Position' record/3
Number of Conductors	TBD
Number of Grids	Number of records with data item names of form 'Grid_####'
Number of Species	Number of records with data item names of form 'Species_##'
Number of Materials	Number of records with data item names of form 'Material_####'
Number of Pages	Number of records with data item names of form 'Prcls_S####_****' where #### is the species number.
Number of ParticlesOnPage	Length of 'Prcls_S##_****' where ## is the species number and **** is the page number.

6.1.10 Other

If the time value is “-1”, the data is treated as time independent.

N2kDB maintains a hard-coded table of the few data items that go in files other than the main one.

6.2 N2kDB Internal Structure

The **N2kDB** dynamic library consists of an API (application programmer interface), **N2kDB**, and five classes. **N2kDB** is the interface between the Java, Fortran, and C++ code of *Nascap-2k*, and the C++ database code. Interfaces to all the functions of Table 3 are provided from Fortran, C++, and Java. (Except that the four “Grid” functions are only called from Fortran, and the “AllocateMemory” functions are *not* called from Java.) It provides all the necessary wrappers between *Nascap-2k* and the C++ database code, so the rest of the database code does not need to accommodate the other languages and *Nascap-2k* does not need to accommodate the specific needs of the database code. For example, character data coming from Fortran, such as data item names, needs to include both the character string and the length of the string. **N2kDB** converts this information into a C++ string. Calls from Java are through JNI. **N2kDB** initializes **MemoryManager** and then interface with **N2kDBDataModel** and **MemoryManager** methods.

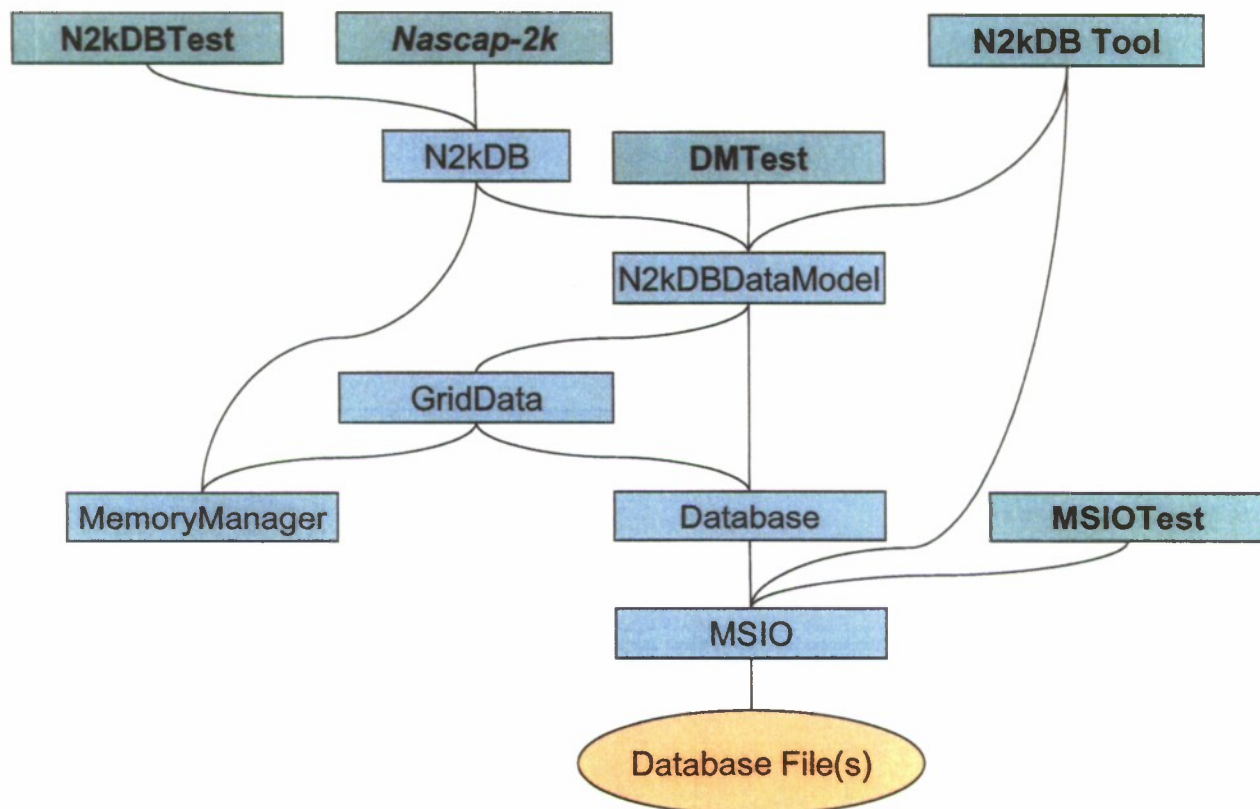


Figure 3. Components of N2kDB in relation to the database file, *Nascap-2k*, and the supporting tools N2kDB Tool and N2kDBTest. DMTest and MSIOTest contain the unit tests of functionality. The lines show the data flow. Data moves down in write operations and up in read operations.

The Database, MSIO, and MemoryManager classes can all be used independently of the components above them in Figure 3.

N2kDBDataModel takes requests from N2kDB and routes them to the correct database components. For example, a request for grid data involves obtaining grid metadata from the GridData class, allocating memory via the MemoryManager class, and finally data retrieval from the Database class. Requests for other data, however, only involve calls to the Database class. At a high level, N2kDBDataModel opens the Database and initializes any other components requiring initialization. The memory manager is automatically used for grid data. (While it is presently contemplated that the memory manager is instantiated every time a database is opened, the database code does not require it.) At destruction, it closes the database and cleans up. DMTest contains the unit tests of the components of N2kDBDataModel.

All classes perform data reads and writes using void pointers. The calling code (N2kDB) is responsible for typecasting the data into appropriate data-types.

The MemoryManager class is a separate class within N2kDB. MemoryManager is implemented as a singleton class, so that there is only one instance and it is accessible from N2kDB and GridData. *Software Requirements Specification for Nascap-2k Database and Memory Manager*

specifies that the memory manager not be a part of **N2kDB**. The separation of the `MemoryManager` class from the `N2kDBDataModel` and `Database` classes adequately addresses this requirement. The requirements of Section 3.3 of the specification are addressed by the `MemoryManager` class.

The `Database` class provides public interfaces to the internal structure of the database. It provides the more user friendly methods (compared with `MSIO`) for manipulating the contents of the database file.

The `GridData` class knows the grid structure and handles reading, writing, and memory requests. The `Database` class instantiates the `GridData` class.

The `MSIO` class controls reading from and writing to the database and the `Database` class provides the only access to its methods. Its component classes are described in Section 6.2.7 below. `MSIOTest` contains the unit tests of the components of `MSIO`.

Table 5. Classes and structs of N2kDB.

Classes	
N2kDBDataModel	N2kDBDataModel class
DMError	Error class for N2kDBDataModel
Database	Database class
GridData	GridData class
MemoryManager	MemoryManager class
MSIO::Header	An MSIO file's header. A Header consists of some metadata and a series of index values (location, length pairs). A header's size can be set when an MSIO file is first created, but it can't be changed afterward
MSIO::Index	Index is a convenience for Header's internal use, which is why it's private with Header as a friend
MSIO::MetaData	Descriptive data placed at the beginning of an MSIO file
MSIO::MSIO	Class for creating, reading from, and writing to MSIO files
MSIO::MSIOError	Error codes for the MSIO class
Structs	
DatabaseConfig	Configuration information for a Database. Presently, this is used to specify the index sizes of the different types of files
GridStructure	Structure that assists in grid bookkeeping by keeping a particular grid's x, y, and z parameters grouped together. It is used by GridData in a std::map to associate a grid number with its structure information.
KeyDataSet< T >	Test structure containing a string key and an array of data associated with the key. (Used for testing only.)
KeyTimeDataSet< T >	Test structure containing a {string, int} key and an array of data associated with the key. This is used for time files. (Used for testing only.)
MSIO::NameNumKey	struct for the purpose of placing a single object into a std::map
ParticleFiles< T >	Since there can be between 0 and 99 particle files, this struct associates particle {key, data} sets with specific particle files.
testStruct	Used to automate the test process.
TimeKey	Allows Database to treat {name, num} keys (time dependent keys) as a single unit, when necessary

6.2.1 Error Handling

The lowest level routines, with single failure modes, return a Boolean indicating success of the operation. Higher level routines, with multiple failure modes, return an integer indicating the error. Each class has its own error list as an enumeration. Utility functions that convert the error enumeration value of one class to the error enumeration of the calling class exist for all classes. In some cases (`MSIO` and `N2kDBDataModel`), a separate class is used to hold the error enumeration and its utility functions.

6.2.2 MemoryManager Class

The `MemoryManager` takes memory requests from the calling code and allocates memory. A pointer to the memory is returned to the caller through an output `void**` parameter. Each request always allocates more memory; memory is not reused. When the `MemoryManager` receives requests, it stores the pointer to the allocated memory in an internal vector. When the `MemoryManager` is destroyed via its `deleteInstance` function, or when its function `freeAllMemory` is called, the `MemoryManager` loops through its vector of memory pointers and frees all of the memory it has allocated. Calling code may also call the `MemoryManager`'s `freeMemory` function, which takes a memory pointer as an argument, to request that a particular allocation be freed. The `MemoryManager` class is data type agnostic. Memory requests are in units of words.

Even though the `MemoryManager` class is initialized by `N2kDBDataModel`, the `GridData` class directs requests to the `MemoryManager`. `MemoryManager` uses the Singleton design pattern, which ensures that a class has only one instance and provides a global point of access to that instance.

Table 6. Public methods of the `MemoryManager` class.

Methods	
static <code>MemoryManager * getInstance ()</code>	Returns to the caller the single instance of the <code>MemoryManager</code> . Creates this instance if it does not already exist.
static int <code>deleteInstance()</code>	Deletes the single instance of the <code>MemoryManager</code> .
static int <code>allocateMemory (void **callerPtrAddress, size_t nwords)</code>	Allocates the amount of memory requested by the caller.
static int <code>freeAllMemory()</code>	Frees all memory allocated by the <code>MemoryManager</code> .
static int <code>freeMemory (void *callerPtrAddress)</code>	Frees one piece of memory allocated by the <code>MemoryManager</code> .

6.2.3 N2kDBDataModel Class

`N2kDB` provides wrappers to `N2kDBDataModel`, so that most of the public methods of `N2kDBDataModel` correspond to **N2kDB** functions. There are three methods that retrieve the last error code, the last error message, and translate an error code to a message. The additional public

methods are used by **N2kDB Tool**. The public methods are given in Table 7. Selected private methods are given in and the error codes are listed in Table 9.

6.2.3.1 Open Database

`N2kDBDataModel` opens the database and initializes the memory manager.

The database is opened by the instantiation of the `Database` class. The `Database` object is allocated with a new and deleted with a delete to insure that the existing object is completely destroyed. Before a database is opened, any database that is already opened is closed and its corresponding `Database` object deleted.

If a database that does not yet exist is opened, a new one is created. No new file is created until data is written into the new database.

`N2kDBDataModel` reads and verifies the index.

`N2kDBDataModel` confirms compatibility of **N2kDB** and of database file(s) with the version string provided by calling code. The version string identifies the calling code as well as including an incremental number.

Much of the functionality of this function is handled by the `Database` class.

6.2.3.2 Close Database

`CloseDatabase` closes the database.

`CloseDatabase` releases all allocated memory.

In general, the index is *not* written by `CloseDatabase`.

6.2.3.3 Read and Write

Data is stored using name keys and hybrid keys. The hybrid keys consist of a name and an integer time index. For convenience, these hybrid keys are referred to as “time” keys. If a read or write function includes “time” as an argument, a hybrid, or “time” key is used.

6.2.3.4 Other

`N2kDBDataModel` does not request data from specific files. Rather, it requests a particular data item name or a particular type of data, and `Database` maps the data item name to a file and a name or hybrid key and retrieves the data.

6.2.3.5 Material, Species, and Grid functions

The `DeleteAllGrids` function deletes the matrix elements (data item names `MtrxElems_#####` and `BndSrfcs_#####`) as well as the grid structure for all the grids.

`ReadGridData(number, *data)` and `WriteGridData(number, *data)` are equivalent to `ReadDatabase(Grid_number, 20, -1, *data)` and `WriteDatabase(Grid_number, 20, -1, *data)`, except that `WriteGridData` updates the grid structure array in the `GridData` object.

`WriteMaterialData(name, int* nwords, *data)` is the same as

- If there is a ### such that the first 80 bytes of the record `Material_###` are the same as name, replace the rest of the bytes with the contents of `*data`.
- If not, add a new record of the form `Material_###` with name in the first 80 bytes and the contents of `*data` in the next `nwords` words.

The function `ReadMaterialData(name, int* nwords, *data)` locates the record `Material_###` such that the first 80 bytes of the record are the name and returns the rest of the record to data.

`WriteSpeciesData(name, real* charge, real* mass)` is the same as

- If there is a ### such that the first 80 bytes of the record `Species_###` are the same as name, write data (the charge, mass, and fraction) into the following words.
- If not, add a new record of the form `Species_###` with contents name in the first 80 bytes, with data (the charge, mass, and fraction) in the following words.

The function `ReadSpeciesData(name, real* data)` locates the record `Species_###` such that the first 80 bytes of the record are the name. The following three words are the charge, the mass, and the fraction.

`GetMaterials` returns an array with contents the first 80 bytes (as characters) of the records `Material_###`.

`GetSpecies` returns an array with contents the first 80 bytes (as characters) of the records `Species_###`.

6.2.3.6 Grid Functions

The `AllocateMemoryGrid`, `ReadGridDatabase`, and `WriteGridDatabase` methods are interfaces to the `GridData` class.

6.2.3.7 Sizes

The `GetLength` and `GetCount` functions are used to get the number of various items. The `GetCount` function gets the number of items with data item names that start with the specified string.

6.2.4 History Function

`ReadHistory` returns time histories of the specified keyword for a specified list of surfaces. The surface number is the Fortran number. The first one is "1".

6.2.4.1 Data Items

`N2kDBDataModel` has functions that return all the data item names and the associated record lengths for use by `N2kDB Tool`.

Table 7. Public methods of `N2kDBDataModel` class.

Method	Functionality
<code>int AllocateMemoryGrid (int dimension, int gridNum, void **data)</code>	Allocates memory for data associated with grid nodes or elements of the specified grid. This is an interface to <code>GridData::AllocateMemory</code> .
<code>int ClearDeadSpace ()</code>	Compacts all files associated with a Database. This is an interface to <code>Database::ClearDeadSpace</code> .
<code>int CloseDatabase ()</code>	Closes the currently open Database. This also instructs the <code>MemoryManager</code> to free all memory.
<code>int DeleteAllGrids ()</code>	Instructs the <code>GridData</code> class to delete all grid data. This is an interface to <code>GridData::DeleteAllGrids</code> .
<code>int DeleteItem (string name, string typeOfItem)</code>	Searches the database for a key of format <i>typeOfItem_###</i> (where <i>typeOfItem</i> is Material, Species, or another string) whose first 80 bytes of associated data contain name and then deletes the record.
<code>int GetCount (string key, int &count)</code>	Retrieves the number of database keys beginning with a prefix. This is an interface to <code>Database::GetCount</code> .
<code>int GetFilenames (vector< string > &files)</code>	Retrieves a list of all filenames associated with the open Database. This is an interface to <code>Database::GetFilenames</code> .
<code>int GetKeys (vector< string > &keys)</code>	Get all of the name keys from the dbdatabase. This does not return the time keys. This is an interface to <code>Database::GetKeys</code> .
<code>int GetKeys (vector< string > &keys, vector< string > &timeStringKeys, vector< int > &timeIntKeys)</code>	Retrieves all of the keys from the dbdatabase. This is an interface to <code>Database::GetKeys</code> .
<code>int GetKeys (vector< string > &keyStrings, vector< int > &keyInts)</code>	Retrieves the time keys from the dbdatabase. This is an interface to <code>Database::GetKeys</code> .
<code>int GetLastError ()</code>	Retrieves the most recent error code that was set.
<code>string GetErrorMessage (int code)</code>	Translates an error code to its string.
<code>string GetLastErrorMessage ()</code>	Retrieves the most recent error message that was set. If the function that set the error message also included additional information (that is appended to the the default message for its code), that information is returned by this function, as well.

Table 7. Public methods of N2kDBDataModel class. (cont.)

Method	Functionality
int GetLength (string key, int &length)	Returns the length in words of the data associated with a name key. This is an interface to Database::GetLength.
int GetLength (string key,int time, int &length)	Returns the length in words of the data associated with a time key. This is an interface to Database::GetLength.
int GetMaterials (int& maxMaterials, vector< string>& keysFound)	Returns a vector containing the first 80 bytes of the data for all the keys of the form "Material_###".
int GetItem(string typeOfItem int& maxItems, vector< string>& itemNames)	Returns a vector containing the first 80 bytes of the data for all the keys of the form <i>typeOfItem_###</i> , where <i>typeOfItem</i> can be Material, Species, or another string.
string GetVersionString ()	Returns the version string of the open Database.
int OpenDatabase (string prefix, string version)	Opens a Database. The following actions are performed. (1) The Database is opened via Database::OpenDatabase. (2) The MemoryManager is initialized. (3) The GridData class is initialized.
int ReadDatabase (string key, int &nwords, void *data, int maxwords)	Reads the data associated with a name key. The caller must pre-allocate the data buffer. This is an interface to Database::ReadDatabase.
int ReadDatabase (string key, int time, int &nwords, void *data, int maxwords)	Reads the data associated with a time key. The caller must pre-allocate the data buffer. This is an interface to Database::ReadDatabase.
int ReadDatabaseGrid (string name, int dimension, int grid, int time, void *data)	Reads grid data data associated with grid nodes or elements of the specified grid from the Database. Also allocates memory. This is an interface to GridData::ReadDatabaseGrid.
int ReadDatabaseGrid (string name, int dimension, int grid, void **data)	Reads data associated with grid nodes or elements of the specified grid from the Database. Also allocates memory. This is an interface to GridData::ReadDatabaseGrid.
int ReadGridData (int number, int nwords, void *data)	Read a grid structure from the database. N2kDBDataModel takes the grid number and calls GridData::ReadGridData. GridData in turn constructs the key "Grid_###" and reads that information from the database The caller must pre-allocate a buffer to store the grid structure.
int ReadHistory (string key, int *entry, int nentries, int &ntimes, void *data, int maxwords)	Retrieves the value of the specified entries associated with the specified "key" at each timestep.
int ReadItemData (string name, string typeOfItem, int nwords, void* data, int maxwords)	Searches the database for a key of format <i>typeOfItem_###</i> (where <i>typeOfItem</i> can be Material, Species, or another string) for which the first 80 bytes of associated data contain name. The following words are then returned in "data." Space for data is allocated by the calling program.
int WriteDatabase (string key, int nwords, void *data)	Writes nwords of data to the database at the location given by a name key. This is an interface to Database::WriteDatabase.

Table 7. Public methods of N2kDBDataModel class. (cont.)

Method	Functionality
int WriteDatabase (string key, int time, int nwords, void *data)	Writes nwords of data to the database at the location given by a time key. This is an interface to Database::WriteDatabase.
int WriteDatabaseGrid (string name, int dimension, int grid, int time, void *data)	Writes data associated with grid nodes or elements of the specified grid to the Database, for a time key. This is an interface to GridData::WriteDatabaseGrid.
int WriteDatabaseGrid (string name, int dimension, int grid, void *data)	Writes grid data associated with grid nodes or elements of the specified grid to the Database, for a name key. This is an interface to GridData::WriteDatabaseGrid.
int WriteGridData (int gridNumber, int nwords, void *data)	Writes grid structure information to the Database. This is an interface to GridData::WriteGridData. Grids must be written in sequence.
int WriteItemData (string name, string nameOfItem, int nwords, void* data)	Write a record with name in the first 80 bytes and data in the following words. If there is a key of format <i>typeOfItem_###</i> (where <i>typeOfItem</i> is Material, Species, or another string) in the database whose first 80 bytes of associated data contain name, use the existing key. If not, use a new key of format <i>typeOfItem_###</i> ..

Table 8. Selected private methods of N2kDBDataModel class.

Method	Functionality
int convertOpenDatabaseError(int dbCode, bool setError = true)	Takes a Database error code (DLError) generated by OpenDatabase and converts it to an N2kDBDataModel error code (DLError).
int convertReadDatabaseError(int dbCode, bool setError = true)	Takes a Database error code (DLError) generated by ReadDatabase and converts it to an N2kDBDataModel error code (DLError).
int convertReadDatabaseGridError(int gridCode, bool setError = true)	Takes a GridData error code (GridError) generated by ReadDatabaseGrid and converts it to an N2kDBDataModel error code
int convertReadGridDataError(int gridCode, bool setError = true)	Takes a GridData error code (GridError) generated by * ReadGridData and converts it to an N2kDBDataModel error code * (DLError).
int convertWriteDatabaseError(int dbCode, bool setError = true)	Takes a Database error code (DLError) generated by * WriteDatabase and converts it to an N2kDBDataModel error code (DLError).

Table 8. Selected private methods of N2kDBDataModel class. (cont.)

Method	Functionality
int convertWriteDatabaseGridError(int gridCode, bool setError = true)	Takes a GridData error code (GridError) generated by * WriteDatabaseGrid and converts it to an N2kDBDataModel error code * (DMError).
int convertWriteGridDataError(int gridCode, bool setError = true)	Takes a GridData error code (GridError) generated by * WriteGridData and converts it to an N2kDBDataModel error code * (DMError).

6.2.4.2 Errors

The error codes returned from N2kDBDataModel are listed in Table 9.

Table 9. Error codes.

OK,
CANT_OPEN,
CANT_CLOSE,
DB_OPEN_NO_GENERAL_FILE
CANT_ALLOCATE_MEMORY,
CANT_ALLOCATE_MEMORY_GRID,
CANT_DELETE_GRID_DATA_KEY,
CANT_DELETE_GRID_KEY,
CANT_FIND_GRID,
CANT_INIT_GRID_DATA,
CANT_INIT_MEMORY_MANAGER,
CANT_READ, // Generic error
DB_MSIO_FILE_MISMATCH
DB_WRONG_KEY_TYPE
DB_CANT_OPEN_FILE
DB_CANT_OPEN_WRITE
DB_CANT_READ_NDB_KEYS
DB_CANT_READ_NPT_KEYS
DB_CANT_READ_NTM_KEYS
DB_CANT_READ_NSE_KEYS
CANT_READ_GRID_STRUCTURE,
CANT_READ_GRID_DATABASE,
CANT_RELEASE_GRID_MEMORY,
CANT_WRITE, // Generic error
CANT_WRITE_GRID_DATABASE,

Table 9. Error codes. (cont.)

CANT_WRITE_GRID_IN_USE,
CANT_WRITE_GRID_OUT_OF_SEQUENCE,
CANT_WRITE_GRID_STRUCTURE, // Generic error
CANT_WRITE_PARTICLE_FILE,
CANT_WRITE_VERSION_KEY,
DB_NOT_OPEN,
DB_READ_ONLY,
DEAD_SPACE_ERROR,
DELETE_ALL_GRIDS_ERROR,
EMPTY_KEY,
EMPTY_DATA,
KEY_NOT_FOUND,
NO_DATA_BUFFER,
DM_DATA_BUFFER_TOO_SMALL
DM_DATA_BUFFER_TOO_LARGE
NOT_OPEN,
DM_UNKNOWN_ERROR
DM_NUM_OF_ERRORS
DM_RECORD_NOT_FOUND
DM_RECORD_TYPE_NOT_FOUND

6.2.5 Database Class

The Database class provides public interfaces to the internal structure of the database. For example, the database as a whole consists of up to four types of files (*prefix.NDB*, *prefix.NSE*, *prefix.NPT##*, and *prefix.NTM*), but this information is unknown to the rest of the system. When `N2kDBDataModel` calls Database's `OpenDatabase` method, Database receives a "prefix" string, which is the name of the database, and a version string that identifies the calling code and a database version number. The Database class then performs the following steps:

1. Search for appropriately named "prefix.suffix" files located in its current directory.
2. For existing databases, performs integrity checking on the database files. All Databases require a *prefix.NDB* file, and this file must have a version number string.
3. For all database files it finds, retrieve the name keys from the files and create a mapping from data item names to database files. This simplifies database reads.
4. Perform integrity checking on the keys. Keys that are in the wrong file are ignored.
5. Confirm compatibility with MSIO version number for all files of database.

6. Reads the database version number from each file of the database and confirm compatibility with **N2kDB** and calling code version number.

Database specifies the file names of the files that comprise the database. It associates certain data item names with certain files and specifies the MSIO file type of each file.

The database version is stored in the main database file. The data item name is DB_Version and the value is of the form AAA.XX.YY.ZZ, where AAA specifies the calling code (“NAS” for *Nascap-2k*).

It is contemplated that at some future time the Database class will be split into two classes, a base class (Database) and a derived class (N2kDatabase). The base class will provide the generic database functions (e.g. talking to **MSIO** files), and the derived class will have all the *Nascap-2k* specific knowledge (e.g. which file contains which data items).

The Database class performs reads and writes on data using void pointers.

Table 10. Public Database class methods.

Method	Functionality
int OpenDatabase (string prefix, string version, bool readOnly=false)	Opens a database. Searches for all files named “prefix.suffix”, where suffix is all suffixes associated with Database files. An existing database must contain a general database file (.NDB), and this file must have a version key. If Database files are found but there is no general file, this is an error. If the general file does not have a version key, this is an error.
int OpenDatabase (string prefix, string version, bool readOnly, const DatabaseConfig &dc)	Open a database and specify some configuration information, such as the maximum number of keys. This only applies to new DBs.
int GetVersionNumber ()	Returns the version number as an integer. For example, a version of 1.2.3 would be returned as 10203. A version of 11.22.33 would be returned as 112233.
string GetVersionString ()	Returns the database version string.
bool IsOpen ()	
int ReadDatabase (string name, int &lengthRead, void *data, int maxWords)	Reads the data associated with a name key. The caller must pre-allocate the data buffer.
int ReadDatabase (string name, int time, int &lengthRead, void *data, int maxWords)	Reads the data associated with a time key (name-number key). The caller must pre-allocate the data buffer.
int WriteDatabase (string name, int words, void *data)	Writes the data associated with a name key.
int WriteDatabase (string name, int time, int words, void *data)	Writes the data associated with time key (name-number key) to the database.
int DeleteRecord (string name)	Removes a name key from the database.

Table 10. Public Database class methods. (cont.)

Method	Functionality
int DeleteRecord (string name, int time)	Removes a time key from the database.
int FindKeys (string keyPrefix, vector< string > &keysFound)	Retrieves the name keys beginning with a prefix.
int GetKeys (vector< string > &keys)	Retrieves all name keys from the database.
int GetKeys (vector< string > &nameKeys, vector< int > &numKeys)	Retrieve the time keys from the database. Each 'i' index of nameKeys and numKeys correspond to each other.
int GetKeys (vector< string > &keys, &timeKeyNames, vector< int > &timeKeyNums)	Retrieves all keys from the database. Each entry of timeKeyNames and timeKeyNums correspond to each other, which could produce multiple entries in timeKeyNames. For example, if the key pairs {"AAA", 1}, {"AAA", 2}, and {"BBB", 3} were inserted, timeKeyNames would contain {"AAA", "AAA", "BBB"} and timeKeyNums would contain {1, 2, 3}.
int GetLength (string name, int& length)	Get length in words of the data associated with a name key.
int GetLength (string name, int time, int& length)	Get length in words of the data associated with a time key.
int GetCount (string name)	Returns number of name keys that start with the specified value.
int GetFileNames (vector< string > &files)	Get the names of the MSIO files associated with the open database.
int CloseDatabase ()	Closes the Database. Member variables are cleared/emptied.
int ClearDeadSpace ()	Removes dead space from all Database files. Calls MSIO::compactMS.
string GetErrorMessage (int code)	Translates an error code to its string.
int GetLastError ()	Retrieves the most recent error code that was set.
string GetLastErrorMessage ()	Retrieves the most recent error message that was set. If the function that set the error message also included additional information (that is appended to the the default message for its code), that information is returned by this function, as well.

Table 11. Selected private Database class methods.

Method	Functionality
Bool readNptKeys ()	Searches the prefix.NPT## files (particles) and places the keys in the keyFileMap.
Bool readNseKeys ()	Searches the prefix.NSE files (special elements) and places the keys in the keyFileMap.
Bool readNtmKeys ()	Searches the prefix.NTM files (time dependent quantities) and places the keys in the keyFileMap.
Bool readNdbKeys ()	Searches the prefix.NDB files (general) and places the keys in the keyFileMap.
bool isNptKey (string key)	Determines whether a key is a particle key
bool isNseKey (string key)	Determines whether a key is a special element key
bool isNdbtKey (string key)	Determines whether a key is not a particle or special element key.
Int checkDatabaseFiles ()	When a database is opened, this function confirms that all files are compatible with each other.
MSIO::MSIO* findMsioFile (string key, uint32 time = 0)	Searches the keyFileMap (and timeKeyFileMap) and returns a pointer to the MSIO file it's in. If the key is not in a file, NULL is returned. This is to reduce the number of (relatively slow) disk accesses.
MSIO::MSIO* findParticleFile (string name)	Map a species name to a specific particle file.
void getMsioFiles (vector<MSIO::MSIO*>& msioFiles)	Retrieve the names of the MSIO files that Database is using.
int getParticleFileNumber (string particleFileName)	Parses the name of the particle file and returns the numeric part of the filename extension. For example, if the particle file were named "prefix.NPT04", this function would return 4.
int getParticleSpeciesNumber (string particleKey)	Parses the name of the particle species (key) and returns the species number. For example, given a key "Prcls_S01_2345", this function would return 1.
bool parseVersionString ()	Parse the version string from the general database file.
int readVersionKey ()	Reads the version key from the general database file.
void setVersionString ()	Initializes the versionString member with the default string.
int writeVersionString ()	Writes the versionString to the general database file.

6.2.6 GridData Class

The class GridData is used by the ReadDatabaseGrid, WriteDatabaseGrid, and AllocateMemoryGrid functions of N2kDB. These functions read, write, and allocate memory for data associated with grid nodes or elements.

GridData is responsible for maintaining grid structure information, such as how many grids there are and how big each grid is. The information is updated whenever the database is opened or the grid structure information is saved by WriteGridData

If the grid structure is changed after memory for gridded data is allocated, the possible error is flagged.

GridData retrieves information on the grid structure from the database. The grid structure is read from and written to the database using the keyword Grid_####. The grid structure is stored in Fortran common blocks. The first word is the grid number and the following three words give the dimensions of the grid in the x, y, and z directions.

The number of items in a grid is $nx*ny*nz$. So the number of words to be read/written/allocated by ReadDatabaseGrid/WriteDatabaseGrid/AllocateMemoryGrid is $dimension*nx*ny*nz$, where dimension is in the argument list.

On request from N2kDBDataModel, GridData makes a memory request from the MemoryManager, and then a pointer to the MemoryManager's data buffer is passed to the Database class, which handles the actual reading.

GridData also constructs appropriate MSIO name keys from Nascap-2k data item names.

Table 12. Public methods of GridData class.

Method	Functionality
int AllocateMemory (int &dimension, int *grid, void *data)	Uses the MemoryManager to allocate memory for data associated with grid nodes or elements of the specified grid.. The number of words allocated is the dimension multiplied by the x, y, and z values for the requested grid number.
int Init (Database *database)	Initializes the GridData class. Gets an instance to the MemoryManager. Queries the database for existing grid information. Grid numbers must be sequential. This function must be called prior to performing any grid operations.
int ReadGridData (int *grid, int nwords, void *data)	Reads the grid structure of a particular grid. This function does NOT allocate memory: It is allocated by the caller.
int WriteGridData (int *grid, int nwords, void *data)	Writes the grid structure of a particular grid.
ReadDatabaseGrid (string name, int dimension, int grid, int time, void **data)	Reads data associated with grid nodes or elements of the specified grid. This function allocates memory via the MemoryManager. The amount of memory allocated is $dimension * grid.x * grid.y * grid.z$. This function interfaces with the Database class. It constructs a Database key by combining the name and grid number.
ReadDatabaseGrid (string name, int dimension, int grid, void **data)	Reads data associated with grid nodes or elements of the specified grid. This function allocates memory via the MemoryManager. The amount of memory allocated is $dimension * grid.x * grid.y * grid.z$. This function interfaces with the Database class. It constructs a Database key by combining the name and grid number.

Table 12. Public methods of GridData class. (cont.)

Method	Functionality
WriteDatabaseGrid (string name, int dimension, int grid, int time, void *data)	Writes data associated with grid nodes or elements of the specified grid. This function allocates memory via the MemoryManager. The amount of memory allocated is dimension * grid.x * grid.y * grid.z. This function interfaces with the Database class. It constructs a Database key by combining the name and grid number.
WriteDatabaseGrid (string name, int dimension, int grid, void *data)	Writes data associated with grid nodes or elements of the specified grid. This function allocates memory via the MemoryManager. The amount of memory allocated is dimension * grid.x * grid.y * grid.z. This function interfaces with the Database class. It constructs a Database key by combining the name and grid number.
int DeleteAllGrids()	Deletes the grid structure and grid-related information (database keys MtrxElms_##### and BndSrfcs_#####) from the database. Releases memory allocated for data associated with grid nodes or elements. Does <i>not</i> delete gridded data from the database as there is no clear algorithm for doing this.
int GetCount(int &count)	Get the number of grids.
int GetGridNumbers (vector<int > &gridNums)	Retrieves the grid numbers.
string GetErrorMessage (int code)	Translates a GridError code to its corresponding string.
int GetLastError ()	Retrieves the most recent GridError code that was set.
string GetLastErrorMessage ()	Retrieves the most recent GridError message that was set. If the function that set the error message also included detailed information (an extra message appended to the the default message for its code), that information is returned by this function, as well.

Table 13. Selected private member functions of GridData class.

Method	Functionality
int AllocateMemory (int dimension, int gridNum, void **data)	
string CreateGridKey (string name, int gridNum)	Create a database key by appending the grid # to the "name" key. For example, grid #1 has a grid key of "GRID_001", grid 10 has a key of "GRID_010", grid 100 has a key of "GRID_100".
int ExtractGridStructure(void* gridBuffer, GridStructure& gridData)	Given a buffer of grid data, pull out the structure information. This is the grid #, X, Y, and Z fields.
GridStructure * FindGrid (int gridNum)	Find the grid structure associated with this grid number. This searches the internal record keeping; it does not re-query the database.
int GetLastGridNum ()	Get the last/highest grid # we have.
int ReadGridStructure (string fullKey, int nwords, void **data)	This function reads a grid structure from the database. This can be used to either retrieve the size of a grid, or the ENTIRE grid structure information (that would be passed to Fortran). For example, if the caller (such as this GridData class) only wanted the grid size, then only the first 4 words would be necessary (nwords = 4 * wordSize). If the caller (such as Fortran code) wanted the entire grid structure, the caller would provide "nwords" and a buffer pointed to by "data" of the appropriate size.
int ReadDatabaseGridAux (string name, int gridNum, int dimension, string &fullKey, int &nwords, void **data)	Provides most of the functionality of ReadDatabaseGrid.
int UpdateGridStructure ()	This updates the internal bookkeeping of grid structure information. Retrieves all grid keys from the Database, and then calls ReadGridStructure to get the latest structure information for each grid.

6.2.7 MSIO Class

MSIO has been rewritten in C++ with expanded functionality. The input and output of the public methods are functionally the same as in the Fortran implementation, except that WritMS writes the actual length of the new record in the index (even if the new record is shorter) and checks to see if there is enough space before the next record to write the new data. This approach allows a record to shorten and expand without necessarily placing the record at the end of the file.

All name keys are converted to upper case before comparisons.

The code optionally translates **MSIO** files of the earlier format to the new format.

At this time, no provision for calling **MSIO** directly from Fortran is made. The code is written so that adding wrappers to accommodate calling from Fortran will be straightforward.

Optional read only access is accommodated, so that **N2kDB Tool** can be used simultaneously with *Nascap-2k*.

The public methods of **MSIO** are given in Table 14.

Table 14. Public methods of MSIO class.

Method	Functionality
int closeMS ()	Closes the MSIO file and resets some internal flags.
int compactMS ()	Compresses an MSIO file to remove gaps between data segments.
int convertFastIO (string oldFname, string newFname=“”)	Converts an old FastIO file into a new MSIO file. Examines the data in the old file and attempts to determine what type of file it is (i.e. name keys, 16-bit record number, or 32-bit record number).
int deleteMS (string key)	Removes a key from an MSIO file.
int deleteMS (int key)	Removes a key from an MSIO file.
int deleteMS (string key1, int key2)	Removes a key from an MSIO file.
int getWords (int key)	Looks up a key in the MSIO file and returns how many words of data that key has.
int getWords (string key)	Looks up a key in the MSIO file and returns how many words of data that key has.
int getWords (string key1, int key2)	Looks up a key in the MSIO file and returns how many words of data that key has.
int getBytes (int key)	Looks up a key in the MSIO file and returns how many bytes of data that key has.
int getBytes (string key)	Looks up a key in the MSIO file and returns how many bytes of data that key has.
int getBytes (string key1, int key2)	Looks up a key in the MSIO file and returns how many bytes of data that key has.
int getIndexSize ()	Returns the maximum number of keys that can go into this MSIO file.
string getFilename ()	Returns the name of this MSIO file.
int getKeys (vector< string > &keyList)	Retrieves all of the name keys in the MSIO file.
int getKeys (vector< int > &keyList)	Retrieves all of the number keys in the MSIO file.

Table 14. Public methods of MSIO class. (cont.)

Method	Functionality
int getKeys (vector< string > &keyList1, vector< int > &keyList2) const	Retrieves all of the hybrid keys in the MSIO file. Retrieves all of the keys in the MSIO file. Each {keyList1[i], keyList2[i]} pair represents a single name-number key. Therefore, if the following keys were written: AAA, 1 AAA, 2 BBB, 3 Then keyList1 would contain {AAA, AAA, BBB} and keyList2 would contain {1, 2, 3}.
MSIOType getKeyType () const	Returns a flag indicating what type of keys this MSIO file will hold (MSIO_STRING, MSIO_INT, MSIO_STRING_INT).
string getErrorMessage (int code)	Translates an error code into its corresponding (default) string.
Int getLastError ()	Returns the last error code set by an MSIO function.
String getLastErrorMessage ()	Returns the error string associated with the last error code set by an MSIO function.
UInt64 getLocationBytes (string key)	Returns the location of the data associated with the key.
UInt64 getLocationBytes (uiint32 key)	Returns the location of the data associated with the key.
UInt64 gctLocationBytes (string key1, int key2)	Returns the location of the data associated with the key.
Int getLocation (string key)	Returns the location of the data associated with the key.
Int getLocation (int key)	Returns the location of the data associated with the key.
Int getLocation (string key1, int key2)	Returns the location of the data associated with the key.
Int getNumKeys ()	Returns the number of items currently in the MSIO file.
Int getSectorSize ()	Returns the sector size associated with this file.
Int getNameKeySize ()	Returns the maximum number of characters that can be in a name key.
Int getVersionNumber()	Returns the version as a number, to do numerical comparisons of versions. Examples: 1.0.3 → 10003 12.1.0 → 120100 1.2.25 → 10225.
String getVersionString ()	Returns the version string.
Bool isFull ()	Returns true if the MSIO file is full.
Int openMS (string fname, MSIOType type, bool readOnly=false)	Opens an MSIO file, with the key type specified. If the file is new, this sets the key type. If the file already exists, then an error is raised if the file's existing type does not match the "type" parameter. If the file does not already exist, an empty header is created and stored in memory. The file will not exist until writeMS is called.
Int openMS (string fname, bool readOnly=false)	Opens an MSIO file. If the file does not already exist, then the key type is the default (name).

Table 14. Public methods of MSIO class. (cont.)

Method	Functionality
int openMS (string fname, MSIOType type, int indexSize, bool readOnly=false)	Opens an MSIO file, with the key type specified. If the file is new, this sets the key type. If the file already exists, then an error is raised if the file's existing type does not match the "type" parameter. If the file does not exist yet, the # of keys in the index is set to indexSize. If the file exists, this parameter is ignored.
int readMS (string key, void *dataBuf, int dataBufSize, int &lengthRead)	Reads a key from an MSIO file; the data buffer MUST be pre-allocated, so the caller should call getWords() prior to calling readMS().
int readMS (string key1, int key2, void *dataBuf, int dataBufSize, int &lengthRead)	Reads a key from an MSIO file; the data buffer MUST be pre-allocated, so the caller should call getWords() prior to calling readMS().
int readMS (int key, void *dataBuf, int dataBufSize, int &lengthRead)	Reads a key from an MSIO file; the data buffer MUST be pre-allocated, so the caller should call getWords() prior to calling readMS().
string sayKeyTypeError (string fname, MSIOType badType)	Returns an error message indicating that the wrong type of key was used in accessing the MSIO file, for example, if the user tries to insert a record number into a name key file. The error message is of this format: "In function <name>: A database file of type int received a key of type string".
int writeMS (string key, void *dataBuf, int nwords)	Writes a key and associated data to an MSIO file.
int writeMS (int key, void *dataBuf, int nwords)	Writes a key and associated data to an MSIO file.
int writeMS (string key1, int key2, void *dataBuf, int nwords)	Writes a key and associated data to an MSIO file.

The structure of the class is shown in Figure 4.

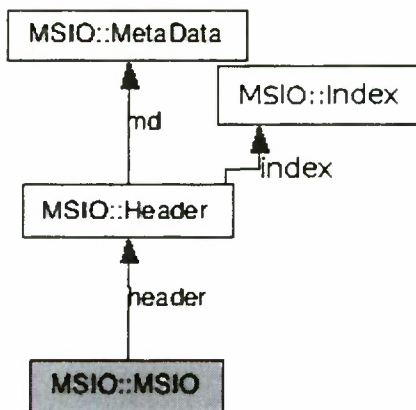


Figure 4. Structure of MSIO class.

The index segment associates a key with its data by mapping the key to a `{location, length}` pair, where *location* corresponds to the data's location in the file in sectors, and *length* is the number of words of data associated with the key. The number of bytes used to specify the location and the length within the header must be the same as in the code. On different operating systems and compilers, an "int" or "long" might have a different numbers of bytes. The number of bytes per word for the data, including the header, is specified in the metadata.). A compiler dependent typedef statement is used to define a type "uint32," an unsigned integer of the appropriate length in bytes. The length and location are then specified to be of type uint32. When the location in the file is needed in bytes, for the actual reading and writing, the type "uint64" is used.

The `Header` object is a private property of `MSIO`, the space for which is allocated by `OpenMS`. The `Index` and `MetaData` objects are private properties of the `Header` object. Any properties of the header (`Index` or `MetaData`) that are needed by outside code interface with `MSIO`, rather than accessing the header directly. Private methods are used for the actual opening, closing, reading, and writing.

The `MSIO` class performs reads and writes on data using void pointers. The calling code is responsible for performing any necessary typecasting. Data reads and writes are performed by the C `stdio.h` library functions `fread()` and `fwrite()`, respectively. The file pointer is moved using `fseek()`.

`MSIO` stores the database keys in memory using a C++ standard library "map" object. Each unique key is mapped to an index structure. Key lookups are performed using the map object's `find` function.

Table 15. Public member functions of MSIO:Header class.

Method	Functionality
int getMaxSize ()	
int getSectorSize ()	Get the sector size of the MSIO file.
int getSize ()	
int getNameKeySize ()	
MSIOType getType () const	
string getVersionString ()	
int getVersionNumber ()	
int getWordSize ()	
void setIndexSize (int indexSize)	
int setType (MSIOType type)	
string sayKeyTypeMessage (MSIOType badType) const	
int erase (string key)	Remove a key from the index.
int erase (int key)	Remove a key from the index.
int erase (NameNumKey key)	Remove a key from the index.
int find (const string key, int &location, int &length)	Find a key in the header.
int find (const int key, int &location, int &length)	Find a key in the header.
int find (const NameNumKey key, int &location, int &length)	Find a key in the header.
int getKeys (vector< string > &keyList) const	Retrieve all of the string keys from the index.
int getKeys (vector< int > &keyList) const	Retrieve all of the int (record #) keys from the index.
int getKeys (vector< NameNumKey > &keyList) const	Retrieve all of the string-int keys from the index.
int insert (string key, uint64 location, int length)	Inserts a record into the header.
int insert (int key, uint64 location, int length)	
int insert (NameNumKey key, uint64 location, int length)	
int update (const string key, uint64 location, int length)	
int update (const int key, uint64 location, int length)	

Table 15. Public member functions of MSIO:Header class. (cont.)

Method	Functionality
int update (const NameNumKey key, uint64 location, int length)	
void clear ()	Erase the data in a header object.
int create (MSIOType type, int indexSize=0)	Create a new MSIO file's header. This just initializes data structures; it does not write it to disk.
int findLength (const uint64 location)	For a given data address in bytes, retrieve its size in words.
uint64 findNextLocation (const uint64 location)	Find the next EMPTY data address AFTER this one. If the location passed in to this function happens to be empty, it still returns the NEXT empty location AFTER this one.
int getSeekAmount ()	
bool isFull ()	
int read (FILE *fp)	Reads the header from MSIO the open file pointed to by fp.
int updateLocation (const uint64 oldLocation, const uint64 newLocation)	This updates a key's location. This searches the header for a key at oldLocation and then updates the record's location to newLocation.
int write (FILE *fp)	Write a header into the open file pointed to by fp. SIDE EFFECTS: The file pointer is moved. This function seeks to the beginning of the file prior to writing the header, and then leaves the pointer at the file position after the header.
static MSIOType readType (std)	
static int sayDefaultIndexSize ()	Usage: int i = Header::sayDefaultIndexSize();.
static int sayDefaultNameKeySize ()	The default name-key size if a new MSIO file were created. Usage: int i = Header::sayDefaultNameKeySize();.
static string sayDefaultVersionString ()	The version string that would be generated if a new MSIO file were created. Usage: string s = Header::sayDefaultVersionString();.
static string sayTypeString (MSIOType type)	Converts an MSIOType into its corresponding string (e.g. "string", "int"). Usage: string s = Header::sayTypeString(type);.

6.2.7.1 MSIO Files

New **MSIO** files consist of a *header* segment and a *data* segment. The *header* segment consists of *metadata* followed by an *index*.

The metadata is 80 bytes (20 4-byte words) and contains the information in Table 16. If any additional data about the database, such as the byte order, would be useful in the future, it can be added using the reserved space in the metadata record.

Table 16. Content of MSIO metadata record.

What	Size	Description
Version String	12 characters (12 bytes)	Format: MSIOXX.YY.ZZ, where XX, YY, and ZZ are digits.
Type of keys in file	1 character (1 byte)	Indicates what type of keys are in the index. Available values are: “1” for name key file, “2” for number key file, and “3” for a hybrid key file.
Size of name keys	1 int (4 bytes)	Length of name keys in bytes (characters).
Index size	1 int (4 bytes)	Maximum number of keys that can be stored in this file.
Sector size	1 int (4 bytes)	Block size of data sectors in bytes.
Word size	1 character (1 byte)	Word/bit size of data, e.g. a “word size” of 4 means it is a 32-bit file.
Reserved	54 bytes	For future use.

N2kDB files have a sector size of 512 bytes.

The index describes where in the file each record is located. The index entry for each record consists of a key, the location (in sectors from the beginning of the file) at which the data starts, and the length of the record (in words). There are three types of keys. For name key files, the key is the ASCII representation of a string, which must be a whole number of words long (multiple of 4 characters for 32 bit words). For number key files, the key is an unsigned integer. For hybrid key files, the key is the ASCII representation of a string followed by an unsigned integer. For hybrid key files, the length of the key is the number words in the string plus one for the integer. For keys of length *length*, the key starts at word $l + (length + 2) * i$, where *l* is the length of the metadata and *i* is an index that starts at “0”. The location and length are in the words immediately following.

6.3 N2kDB Tool

N2kDB Tool provides a user interface into (1) a *Nascap-2k* database, (2) an **MSIO 2.0** file, or (3) an **MSIO** file of the earlier format.

If **N2kDB Tool** cannot open the database as read/write, it opens it as read-only, so that the database can be examined at the same time as it is open with *Nascap-2k*. With some operating systems and compilers, both **N2kDB Tool** and *Nascap-2k* can open the same database at the same time as read-write.

If a *Nascap-2k* database is opened, the content of all files of the database is available. On opening, the file metadata and the index are displayed.

When the user specifies the data item name, data format, and start and end locations, **N2kDB Tool** creates a text window in which the data is displayed. It is possible to copy the contents of this window to the clipboard. When the database is opened in read/write mode, the user is able to edit the contents of the window. There is a save button, which when clicked, saves the contents of the window into the location from which it came. If the data in the window is larger than the specified location, the data is written at the end of the file, as is normal for **MSIO**. If the data in the window is shorter than the specified location, the tail end of the data on the disk is *not* overwritten.

6.3.1 User Interface

Two versions of **N2kDB Tool** have been built. The simplest reads from and writes to the command line. The type of file to be read and the filename are specified using keyword input. The data to be read and its format are specified using keyword input. No provision is made for writing to the database. The files of **N2kDB** are linked in to form a single executable.

The other version of **N2kDB Tool** has a Java user interface to an underlying C++ dynamically linked library that is statically linked to the underlying classes within **N2kDB**. There are two open commands, one for a *Nascap-2k* database and one for a single **MSIO** file. There are two screens. The first screen shows the index and the second the data. The screen that shows the index appears slightly differently depending on if a *Nascap-2k* database or a single **MSIO** file is opened. The screens are shown in Figure 5, Figure 6, and Figure 7. The data window allows for copy and paste.

N2kDBTool

File

MSIO file: C:\SpacePhysics\Nascap2k\N2kDBTest\bin\N2kDBTest.NDB
 Version: MSIO00.00.02
 Index size: 23
 Sector size: 512

Keyword	Start (sectors)	Length (words)
DBLIB_DATAISGRID	471	27
DBLIB_DATALENDIM	472	27
DBLIB_DATANAMES	586	162
DB_VERSION	469	3
GELM_GIS_001	591	4845
GELM_GIS_002	781	2197
GEN_CURRENTTABLE	871	15001
GEN_FLUXTABLE	869	202
GEN_OBJECTSIZE	590	7
GNOD_POT_001	629	19380
GNOD_POT_002	799	8788
GRID_001	588	20
GRID_002	589	20
MATERIALS	473	316
NOD_POSITION	581	168
POT_CONDUCTOR	585	108
SRF_ATTRIBUTE	576	540
SRF_CENTROID	478	162
SRF_ELEMENT	476	216
SRF_F0CHARGEDEN	868	108
SRF_NORMAL	480	162
SRF_POTENTIAL	583	108
SRF_POTFIXED	584	54

Figure 5. Screen that appears when an MSIO file is opened.

File	
Nascap-2k Database: C:\SpacePhysics\Nascap2k\N2kDBTest\bin\N2kDBTest	
2 files: C:\SpacePhysics\Nascap2k\N2kDBTest\bin\N2kDBTest.NDB C:\SpacePhysics\Nascap2k\N2kDBTest\bin\N2kDBTest.NTM	
Database version: NAS.00.00.00	
Keyword	Length (words)
DBLIB_DATAISGRID	27
DBLIB_DATALENDIM	27
DBLIB_DATANAMES	162
GELM_GIS_001	4845
GELM_GIS_002	2197
GEN_CURRENTTABLE	15001
GEN_FLUXTABLE	202
GEN_OBJECTSIZE	7
GNOD_POT_001	19380
GNOD_POT_002	8788
GRID_001	20
GRID_002	20
MATERIALS	316
NOD_POSITION	168
POT_CONDUCTOR	108
SRF_ATTRIBUTE	540
SRF_CENTROID	162
SRF_ELEMENT	216
SRF_FOCHARGEDEN	108
SRF_NORMAL	162
SRF_POTENTIAL	108
SRF_POTFIXED	54
GNOD_POT_001 Timestep 1	19380
GNOD_POT_001 Timestep 2	19380
GNOD_POT_001 Timestep 3	19380
GNOD_POT_002 Timestep 1	8788
GNOD_POT_002 Timestep 2	8788
GNOD_POT_002 Timestep 3	8788
POT_BIAS_SURF Timestep 0	8788
SRF_POTENTIAL Timestep 1	108
SRF_POTENTIAL Timestep 2	108
SRF_POTENTIAL Timestep 3	108

Figure 6. Screen that appears when a *Nascap-2k* database is opened

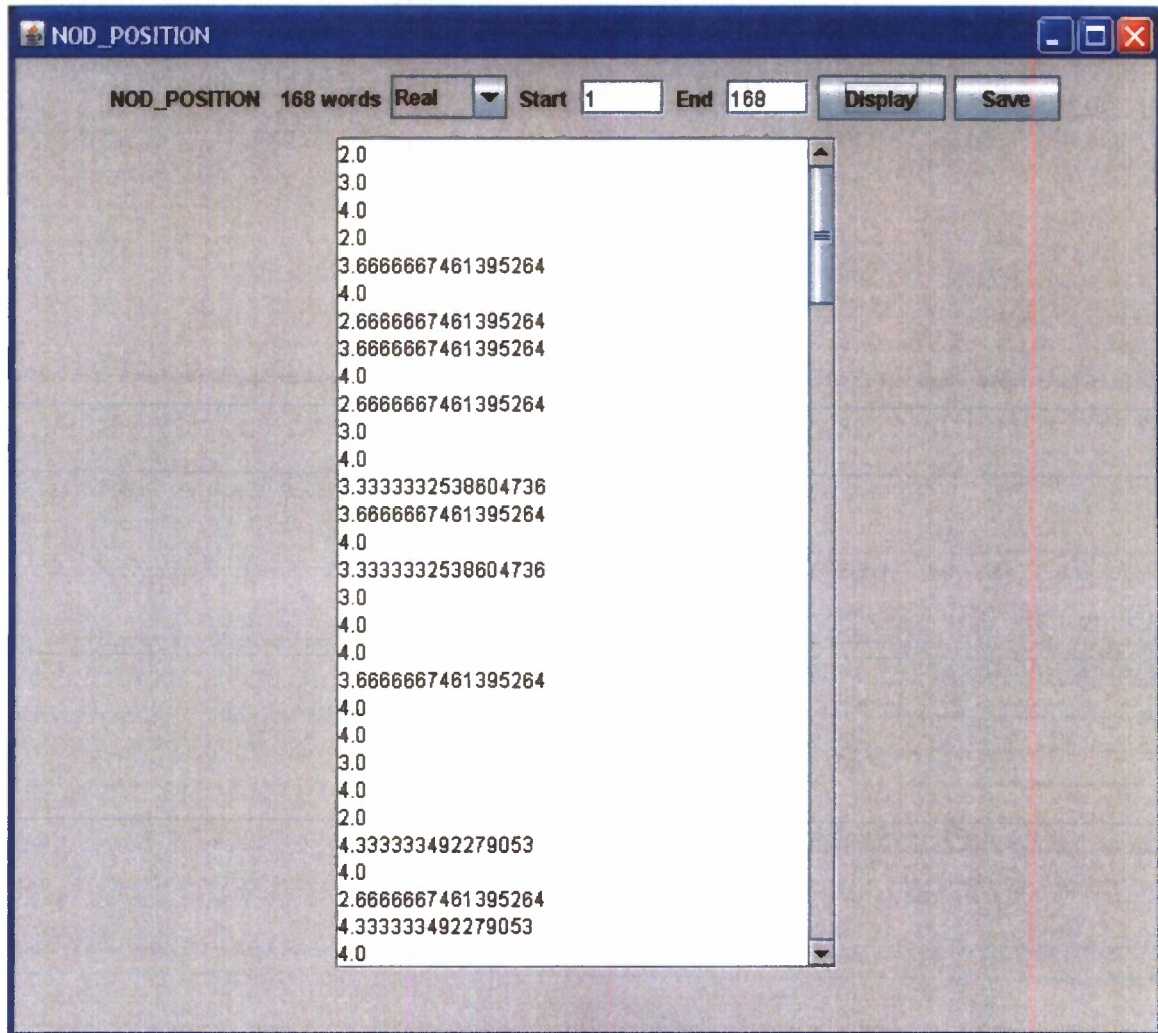


Figure 7. Screen that appears when button on Figure 5 or Figure 6 is clicked.

6.3.2 Code Structure

N2kDB Tool exists in two forms, a single executable and a combination of a Java user interface and a dynamically linked library. The single executable, **N2kDBToolConsole**, consists of a driver and a statically linked library. The dynamically linked library consists of JNI (Java Native Interface) wrappers to the same statically linked library. Both are linked with the underlying statically linked libraries that comprise **N2kDB**.

6.3.2.1 N2kDB Tool library

The library functions used by both user interfaces are given in Table 17.

Table 17. N2kDB Tool Library.

Method	Functionality
void OpenMSIOFile(string filename, vector<string>& nameKeys, vector<int>& numberKeys, vector<int>& recordLength, vector<int>& locationInFile, MSIO::MSIO& file, bool& opened, String& errmsg)	Opens the MSIO file and initializes the vectors containing the keys in the file and their sizes. The first argument is the file name and the rest are return values. The length of the vectors is determined by the type of file (name, number, or name/number pair) Note: An MSIO file will only have one type of key, so only one of the lengths will be nonzero. The record lengths are specified in words and the locations are specified in sectors. The MSIO object is returned for subsequent file access. Opened is set to true or false, depending if the requested file could be accessed. Errmsg is set to a message stating why the MSIO file couldn't be opened.
void OpenN2kDBDataBase(string databaseName, vector<string>& nameKeys, vector<int>& nameKeyRecordLength vector<string>& namePartOfNameNumberKeys, vector<int>& numPartOfNameNumberKeys, vector<int>& nameNumKeyrRecordLength N2kDBDataModel& dataModel, bool& opened, string& errmsg)	Opens the database and returns the list of keys in the database and their sizes. The first argument is the database name and the rest are return values. The record lengths are specified in words. The dataModel object is returned for subsequent database access. Opened is set to true or false, depending if the requested file could be accessed. Errmsg is set to a message stating why the MSIO file couldn't be opened.
bool CheckIsDataBase(string dbName)	Returns true if the string is possibly the name of an N2kDB database.
void N2kDBGetData(N2kDBDataModel& dataModel, string nameKey, int numberKey, int nwords, void* databuf)	Places the first nwords words from the record identified by nameKey, and optionally numberKey, at the memory location databuf.
void N2kDBGetData(MSIO::MSIO& file, string nameKey, int numberKey, int nwords, void* databuf)	Places the first nwords words from the record identified by nameKey, and optionally numberKey, at the memory location databuf.
void N2kDBPutData(N2kDBDataModel& dataModel, string nameKey, int numberKey, int nwords, void* databuf)	Saves nwords words to the record identified by nameKey, and optionally numberKey, starting from the memory location databuf.

Table 17. N2kDB Tool Library. (cont.)

Method	Functionality
void N2kDBPutData(MSIO::MSIO& file, string nameKey, int numberKey, int nwords, void* databuf)	Saves nwords words to the record identified by nameKey, and optionally numberKey, starting from the memory location databuf.
void getNumMSIOKeys(string filename, int& numNameKeys, int& numNumberKeys, int& numNameNumberKeys)	Opens an MSIO file and returns the number of keys.
void getNumDBKeys(string N2kDBprefix, int& numNameKeys, int& numNameIntKeys)	Opens the N2kDB database and returns the number of keys.
void getDBInfo(string N2kDBprefix, string& N2kDBfilelist, string& N2kDBVersion)	Opens an N2kDB database and returns a string containing the names of the files comprising the database and the database version number.
void getMSIOInfo(string filename, string& version, int& sectorSize)	Opens an MSIO file and returns the version number and the sector size.

6.3.2.2 N2kDB Tool Java interface

The Java interface has four classes: N2kDBTool, DataPanel, ButtonColumn, and N2kDBTableModel. N2kDBTool is an extension of the JFrame class and is the entry point. Its methods are given in Table 18. The native methods it uses are given in Table 19.

N2kDBTableModel generates the tables shown in Figure 5 and Figure 6. ButtonColumn turns a column of the table into clickable buttons. DataPanel is an extension of the JFrame class that displays the screen shown in Figure 7. Its methods are shown in Table 20.

Table 18. Methods of N2kDB Tool class of N2kDB Tool Java interface.

Method Name and Arguments	
<code>void msioIndex()</code>	Draws the screen showing all keywords of an MSIO file. The keywords are indexed starting with 0 on the first line • <code>display[]</code> - Array of JButtons. Each button brings up the element info screen for the appropriate keyword. Each button has its own unique action command with the following syntax: <code>keyIndex, keyType, databaseIndex</code>. (For MSIO files, <code>databaseIndex = -1</code>). • <code>keyword[]</code> - String array displaying the keyword name. • <code>locationInFile[]</code> - array of ints indicating start (sectors). • <code>skSize[], ikSize[], sikSize[]</code> - array of ints indicating number of elements (words) associated with each keyword. There are three different arrays corresponding to the three types of keyword: <code>skSize</code> for name only keys, <code>ikSize</code> for number only keys, and <code>sikSize</code> for hybrid keys
<code>void dbIndex()</code>	Draws the screen of all keywords in a database. The keywords are indexed starting with 0 on the first line of the grid. Since a database is made up of individual MSIO files, <code>n2kIndex</code> opens all MSIO files in the database and adds their keywords to the grid in the sequence in which they were read. Each keyword is given a <code>keyIndex</code> and a <code>databaseIndex</code>. <code>keyIndex</code> starts from 0 and runs up to the number of keys in the MSIO file for each MSIO file. <code>databaseIndex</code> starts from 0 and runs up to the number of keys in the entire database. • <code>display[]</code> - Array of JButtons. Each button brings up the element info screen for the appropriate keyword. Each button has its own unique action command with the following syntax: <code>keyIndex, keyType, databaseIndex</code>. • <code>keyword[]</code> - String array displaying the keyword name. • <code>locationInFile[]</code> - array of longs indicating start (sectors). <code>skSize[], ikSize[], sikSize[]</code> - array of ints indicating number of elements (words) associated with each keyword. There are three different arrays corresponding to the three types of keyword: <code>skSize</code> for name only keys, <code>ikSize</code> for number only keys, and <code>sikSize</code> for hybrid keys.
<code>void initArrays()</code>	Uses the info from <code>getMSIOInfo</code> and <code>getDBInfo</code> to create the space that will be needed when calling <code>openN2k</code> and <code>openMSIO</code> .
<code>void initData()</code>	Takes the data returned from <code>openN2k</code> and <code>openMSIO</code> and puts it in the "data" object. The data object is passed to the <code>N2kDBTableModel</code> constructor to create the table that displays database/MSIO file information.
<code>boolean isDB()</code>	Returns the value of the flag specifying whether a database or MSIO file is open.

Table 18. Methods of N2kDB Tool class of N2kDB Tool Java interface. (cont.)

Method Name and Arguments	
int getIKSize(int i)	Returns the number of number keys specified by i.
int getSKSize(int i)	Returns the number of name keys specified by i.
int getSIKSize(int i)	Returns the number of hybrid keys specified by i.
String getNameKeyList(int i)	Returns the name of the name key specified by i.
int getNumberKeyList(int i)	Returns the number of the number key specified by i.
String getNameNameNumKeys(int i)	Returns the name of the hybrid key specified by i.
int getNumNameNumKeys(int i)	Returns the number of the hybrid key specified by i.
actionPerformed(ActionEvent evt)	Responds to user selections.

Table 19. Native methods used by N2kDB Tool Java interface.

Method	Functionality
native getNumMSIOKeys(String fileName, int[] numNameKeys, int[] numNumberKeys, int[] numNameNumberKeys)	Gets the number of name-only keys, number-only keys, and hybrid keys for the MSIO file specified by fileName. Note: An MSIO file will only have one type of key, so only one of these should be nonzero.
native getMSIOInfo(String fileName, String[] version, int[] sectorSize)	Gets MSIO version and sector size for file specified by fileName. version[] and sectorSize[] have length 1. This is a JNI workaround.
native openMSIO(String fileName, String[] nameKeyList, int[] namekeyRecordLength, int[] numberKeyList, int[] numKeyRecordLength, long[] locationInFile, boolean opened, String errmsg)	Reads keys (name, number, hybrid) and number of words for each key from the MSIO file specified by fileName into Java GUI code. The method writes to the parameters it is passed.
native getNumDBKeys(String fileName, int[] numNameKeys, int[] numNameNumberKeys[])	Gets the number of name keys and hybrid keys in the database specified by fileName.
native getDBInfo(String fileName, String dbFileNames[], String dbVersion[])	Gets the names of the files in the database specified by fileName. Gets the version of the database specified by fileName.
native openN2k(String fileName, String[] nameKeyList, int[] nameKeyRecordLength, String[] namePartOfNameNumberKeys, int[] numPartOfNameNumberKeys, int[] numNameKeyRecordLength, boolean opened, String errmsg)	Reads keys (name, number, hybrid) and number of words for each key from Nascap-2k database specified by N2kDBFile into Java GUI code. The method writes to the parameters it is passed.

Table 19. Native methods used by N2kDB Tool Java interface. (cont.)

Method	Functionality
native isDatabase(String fileName)	Returns true if the user requested a Nascap-2k database, and false if the user opened an MSIO file (*.NDB, *.NTM)
native readDatabase(boolean isDB, String name, int number, int size, int[] idata) native readDatabase(boolean isDB, String name, int number, int size, char[] cdata) native readDatabase(boolean isDB, String name, int number, int size, double[] data) native writeDatabase(boolean isDB, String name, int number, int size, int[] idata) native writeDatabase(boolean isDB, String name, int number, int size, char[] cdata) native writeDatabase(boolean isDB, String name, int number, int size, double[] data)	Native methods (N2kDBTool.dll) Reads or writes the elements (words) for the selected key (Note: the selected key is indicated by keyIndex). read/writeDatabase needs to be passed appropriate values for key name, number, and size depending on the key type. name = key names, number = key numbers, size = number of elements (words) for each key. • isDB: identifies the input file as a database or single MSIO file. • name should be an element of nameKeyList[] for name-only keys, -1 for number-only keys, and an element of strNameIntKeys[] for hybrid keys • number should be null for name-only keys, an element of numberKeyList[] for number-only keys, and an element of intNameIntKeys[] for hybrid keys • size should be an element of skSize[] for name-only keys, an element of ikSize[] for number-only keys, and an element of sikSize[] for hybrid keys • data[], idata[], and cdata[] are arrays for "Real" "Integer" and "ASCII" elements, respectively. Note: the number of ASCII elements needs to be 4*size. This is handled manually when the array is declared, when "ASCII" is selected from the combo box, and when read/write database methods are called.

Table 20. Methods of Java DataPanel class of N2kDB Tool Java interface.

Method	Functionality
displayKeyData()	Obtains and displays all the data on the screen.
saveKeyData()	Saves data in text field.
setText(String str)	Sets the data displayed in the text field.
actionPerformed(ActionEvent evt)	Responds to user selections (Data format and Display and Save buttons).
propertyChange(PropertyChangeEvent evt)	Responds to changes in the Start and End fields
JTextArea getElementTextArea()	Returns the text area.

6.3.2.3 N2kDB Tool Console User Interface

After querying the user for the name of the database or file, the code opens the database or file, displays general information about the database or file, and lists the keys. The code then enters a loop that queries the user for the key, displays the record associated with the specified key, and then repeats.

7 TESTING

7.1 Testing of Each Class

Before each class was written one or more testing codes (unit tests) that call all of the public methods under all reasonable circumstances were written. The class must pass the test anew each time it is modified. This approach required a bottom up construction of the new classes. We did *not* use Visual Studio 8's built in unit tests, in order to maintain portability.

7.1.1 Unit Test Code Guidelines

The following guidelines were followed when writing unit tests.

- Test non-trivial public methods and properties (e.g. don't test getter/setter methods).
- Test at least one success case. Test the most typical use of the class/method.
- Check boundary conditions.
- Use negative tests to be sure the code responds to error conditions appropriately. Verify that the code behaves appropriately when it receives invalid or unexpected input values. Verify that it returns errors or throws exceptions when it should.
- Whenever a bug is fixed, add unit tests to verify that the behavior remains fixed.

- Write tests that combine different code modules to implement some of the more complex behaviors of the application.
- Write unit tests that will continue to function as the code changes.
- Reuse creation, manipulation, and assertion code when possible. Don't create instances of classes directly inside a unit test.
- Avoid dependencies between tests. A test will be able to stand on its own. It will not rely on any other test, nor will it depend on tests being run in a specific order. Someone else will be able to take all the tests, run all or just some of them, in any order, and know that they will always behave the same.
- Unit tests will be easy to run in an automated fashion.

7.1.2 MSIO Unit Tests

MSIOTest is an application that tests functionality of the **MSIO** class. It is maintained under version control with the class source code. The **MSIOTest** application can be run in one of two ways:

Mode #1: A specific test case can be listed as a command line argument, e.g.:

MSIOTest.exe 15

MSIOTest.exe 17

MSIOTest.exe 20

Mode #2: If no argument provided, the user is shown a menu of available tests and is prompted to specify one.

In general, each test case creates its own "testXX.bin" MSIO file, where XX corresponds to the name (number) of the test. The one exception is test #23, which converts an old fastio file, fort.20, into a new MSIO file. The test file created by test #23 is called fort_new.20. *NOTE* that fort.20 is not currently part of the project's version control, so it must be manually placed in the current working directory.

While the tests are identified by integers, the numbers are not consecutive.

7.1.2.1 Test 4

Creates, writes to, and reads from and name key file. Creates a key with character, float, and integer data, writes the key to the file, and then reads the key from the file. Performs byte by byte comparison and data comparison of data written to, and read from, the key.

7.1.2.2 Test 5

Tests that read only traits are handled and preserved correctly.

7.1.2.3 Test 6

Writes a name key to a file, prints the seek location. Reads the data from the file and compares it to written data.

7.1.2.4 Test 7

Creates, writes to, and reads from name key file. Creates 2 keys, one with int data and one with double data.

7.1.2.5 Test 11

Tests changing the amount of data associated with a key. First, two records are written. Then, record #2, at the end of the file becomes larger, but does not need to move (as it's at the end of the file). The record at the beginning of the file becomes smaller and therefore does not have to move. This test can't determine whether or not the data elements actually move or not, but as long as we can re-read the keys, then the tests passes.

7.1.2.6 Test 12

In this test, records will look like:

| record #1 | record #2 | record #3 |

Record #2 will increase in size and move to the end. Records then look like

| record #1 | | record #3 | record #2 |

Record #1 will increase in size, but since record #2 was moved, it will *not* need to move. Final result looks like:

| record 1 | | record 3 | record 2 |

This test can't determine whether or not the data elements actually move or not, but as long as we can re-read the keys, then the tests passes.

7.1.2.7 Test 13

Prints some sizeof values.

7.1.2.8 Test 15

This is the same as test12, except that compactMS is called. This test can't determine whether or not the data elements actually move or not, but as long as we can re-read the keys, then the tests passes.

7.1.2.9 Test 15b

This is the same as test 15, but tests with dead space occurring as the result of a deleted record.

7.1.2.10 Test 16

Simple test of listing keys and then erasing one.

7.1.2.11 Test 17

Simple test of writing and re-reading hybrid keys. The keys are

pair #1: {dkey, 12340}

pair #2: {ikey, 56789}

pair #3: {ikey, 6789}

For pairs 2 and 3, the string portions are the same, but the int portions are different, so they are treated as distinct keys.

7.1.2.12 Test 18

Tests search functionality on int keys. Keys numbered 1-9 are placed into the file.

Step 1: Find all keys ≥ 3 .

Step 2: Append " ≤ 7 " to the search, so that we find all keys between 3 and 7.

Step 3: Append " $\neq 6$ " to the search, so that we find all keys between 3 and 7 but not 6.

7.1.2.13 Test 19

Two tests on string-int keys.

Test #1: The string-int keys "dkey" and "ikey" are inserted, and then we search for name keys that contain "ik."

Test #2: One of the string-int keys has an int key > 10000 , and the other has a key < 10000 . The second search appends looking for ints > 10000 to the existing search, so we only obtain keys whose string part contain "ik" and whose int part is > 10000 .

7.1.2.14 Test 23

Convert an old FastIO file, fort.20, into the new MSIO format. In order to truly test success on this test case, we'd need to hard-code the keys in the fort.20 file and check that all of those keys (and ONLY those keys) are in the converted file. Currently, this will return success simply

if the `convertFastIO` routine returns no error AND any kind of key is read from the converted file.

7.1.2.15 Test 32

Confirms that a warning is generated when a buffer passed to `readMS` is smaller than the record's length. Note that the number of words actually read is also returned.

7.1.3 N2kDBDataModel Tests

N2kDBDataModelTest is an application that tests some functionality of the `N2kDBDataModel` class. It is maintained under version control with the class source code.

7.1.3.1 Test #1

A simple test case that is an example of how to interface with `N2kDBDataModel`. It ensures that the appropriate database files are created when keys are written and that the data read from these keys matches the data written to them.

7.1.3.2 Test #2

Tests opening an existing database by creating MSIO files directly and then calling `OpenDatabase`.

7.1.3.3 Test #3

Creates a new database, writes keys and data, and closes the database. Confirms that the keys and data are correctly re-read from the database.

7.1.3.4 Test #4

Creates a new database containing particle keys. As the name of a particle key indicates which particle file it is written into, the MSIO files are then accessed directly to confirm that the particle keys were placed into the correct files.

7.1.3.5 Test #5

Tests Database's `ReadDatabase` function

7.1.3.6 Test #6

Creates grid structure information via the Database's interface, and confirms that the `GridData` class correctly reads the information.

7.1.3.7 Test #7

Confirms that `ReadDatabaseGrid` correctly retrieves grid structure information, calls the `MemoryManager` and then reads data via calls to Database's `ReadDatabase`.

7.1.3.8 Test #8

Confirms that the DeleteAllGrids function deletes the correct keys.

7.1.3.9 Test #9

Tests that the GridData function WriteGridData correctly writes grid structure information.

7.1.3.10 Test #10

Confirms that when a grid is “”, the GridData class will not write new information.

7.1.3.11 Test #11

Like Test #10, confirms that when a grid is “in use”, the GridData class will not write new information

7.1.3.12 Test #12

Confirms that, when GridData is initialized, missing a grid number will raise an error.

7.1.3.13 Test #13

Tests N2kDBDataModel’s WriteDatabaseGrid function

7.1.3.14 Test #14

Confirms that GridData will raise an error if grid numbers are written out of sequence. For example, attempting to write grid numbers 1, 2, 4 will raise an error when grid 4 is written, since there is no grid 3.

7.1.3.15 Test #15

Tests N2kDBDataModel’s ReadGridData function.

7.1.3.16 Test #16

Tests N2kDBDataModel’s ReadDatabaseGrid function.

7.1.3.17 Test #17

Tests N2kDBDataModel’s DeleteAllGrids function.

7.1.3.18 Test #18

Tests N2kDBDataModel’s WriteGridData function.

7.1.3.19 Test #19

Tests N2kDBDataModel's WriteDatabaseGrid function.

7.1.3.20 Test #20

Confirms that an error is written when calling WriteDatabaseGrid for a grid number whose structure is not in the database.

7.1.3.21 Test #21

Confirms that a error is raised if an existing database does not have a general database file (.NDB).

7.1.3.22 Test #22

Confirms that an error is raised if a general database file (.NDB) does not have a database version string.

7.1.3.23 Test #23

Tests that an error is generated if MSIO files are incompatible with each other.

7.1.3.24 Test #24

Tests that the database operates correctly when opened in read only mode.

7.1.3.25 Test #25

Tests reading the time history for a key. Does not check that the key has a time history.

7.1.3.26 Test #26

Verify functionality of ReadItemData, WriteItemData, GetItems, and DeleteItem functions by creating material records, reading them using both ReadItemData and ReadDatabase, deleting records, and subsequently adding records, using names that are an even number of words, that are not, that have spaces, and that are similar to each other.

7.2 Consistency with Specification

We have reviewed this document and *Software Requirements Specification for Nascap-2k Database and Memory Manager* simultaneously and verified that the design satisfies the specification. The manner in which each paragraph of the specification is satisfied is included as Appendix A to this document.

8 INTERMEDIATE IMPLEMENTATION OF N2KDB

Nascap-2k 3.2 uses calls to four subroutines, `dbdata`, `dbinfo`, `dbfile`, and `buffio` to perform all database access and memory management. The initial implementation of the new database and memory management system is transparent to *Nascap-2k* above the level of the database calls, `dbdata`, `dbinfo`, `dbfile`, and `buffio`. These calls were left in the code, as is, but, instead of calling the old database code, function as wrappers to the new code. This simplest implementation and minimum modification of *Nascap-2k* code has been implemented in order to move to the new database with the least disruption. This version of *Nascap-2k* is known as *Nascap-2k* 4.0.#, where # refers to successive versions with the new capabilities of **N2kDB** more fully integrated. The modifications to *Nascap-2k* needed to use the full capabilities of **N2kDB** will be made over the next few months.

The Fortran subroutines `dbdata`, `dbfile`, `dbinfo` all have the same structure. They each consist of calls to three subroutines. The first of these defines constants and handles initialization, the second reads the keyword input and sets the appropriate variables and flags, and the third performs the requested actions. For the interim implementation, the first two of these subroutines were left in place and the third, action subroutine was replaced with a subroutine with the same name that returns pointers to the same locations while using the new database software underneath.

A new version of `buffio` was written that takes the same keyword list argument presently used and returns pointers to the same locations while using the new database software underneath.

In some cases, this temporary structure leads to complicated temporary code. When this is the case, a direct call to **N2kDB** is used.

Before the intermediate implementation of **N2kDB** into *Nascap-2k*, the following changes were made:

- The data item names were replaced with the new names,
- All data initialization that was handled by database commands were replaced by `s3set` and `s3zero` commands.
- The grid data in common blocks was reorganized. The Fortran common blocks containing the grid structure information order the information in three different ways. The decision was made to store the data in the following manner. **N2kDyn** and **Scanner** reformat the data on write and read respectively

`IGrid`, `Nx`, `Ny`, `Nz`, `IParen`, `MsRati`, `M1Rati`, `XmLocl`, `ParOri(3)`, `PriOri(3)`

<code>IGrid</code>	Grid number
<code>Nx,Ny,Nz</code>	# of nodes along grid edges, includes endpoints
<code>IParen</code>	# of parent grid, =0 if this is primary grid
<code>MsRati</code>	Ratio of child/parent mesh units

M1Rati	Ratio of child/primary mesh units
XmLocl	Mesh size of this grid
ParOri(3)	Origin of this grid in parent grid units
PriOri(3)	Origin of this grid in primary grid units

The capabilities of the intermediate implementation are as follows:

- Calls to dbfile replaced with OpenDatabase and CloseDatabase.
- Calls to dbinfo(Inquire GRID) replaced with calls to ReadGridData
- Call to dbinfo(Define GRID) replaced with calls to DeleteAllGrids and WriteGridData
- Calls to dbinfo(Inquire Problem) replaced with calls to GetCount("Grid", ngrids) and ReadGridData as appropriate.
- New version of buffio that uses N2kDB rather than dbdata of DBLIB.
- New version of dtactn that stores the length of the data items in a way that can be retrieved by buffio and ddactn. Other functionality of dtactn is not needed.
- New version of ddactn that uses N2kDB rather than DBLIB.
- N2kDB function GetMaterials is known as GetMaterialsNew.
- The "MatrxElems" and "BndSrfs" data items are saved directly with **N2kDB**.

REFERENCES

- 1 M.J. Mandell, T. Luu, J. Lilley, Analysis of Dynamical Plasma Interactions with High-Voltage Spacecraft, Final Report – Volume II, PL-TR-92-2248 (II), 1992.
- 2 V.A. Davis, M.J. Mandell, S.L. Huston, R.A. Kuharski, B.M. Gardner, Plasma Interactions with Spacecraft, Scientific Report No. 1, AFRL-VS-HA-TR-2007-1062, 2007.
- 3 S. Liang, *Java Native Interface: Programmer's Guide and Specification*, Prentice Hall, 1999.
- 4 http://en.wikipedia.org/wiki/Java_Native_Interface.
- 5 <http://java.sun.com/j2se/1.4.2/docs/guide/jni/>.
- 6 <http://www.ibm.com/developerworks/edu/j-dw-javajni-i.html>.

APPENDIX A. NASCAP-2K 3.2 STRUCTURE

Nascap-2k 3.2 (the last version of *Nascap-2k* to use the old database) is composed of the several components that are listed in Table A1. The relationships between them are shown in Figure A1. The graphical user interface communicates with the computational modules. The computational modules read and write computed data to the database files. The **lapack** dynamically linked library solves matrix equations and **Xerces** understands XML data structures. The **DynaBase** dynamically linked library is a Fortran library and reads quantities from and writes quantities to the database that are needed or generated by the non-Fortran components of *Nascap-2k* (**BEMDLL** or the user interface). **DynaBase** was needed because the *Nascap-2k 3.2* database code is Fortran-centric, so that C++ and Java code cannot call it directly.

All communication with the database is handled by **DBLib**, which also handles memory management. **DBLib** is a static library written in Fortran with the database reads and writes in C.

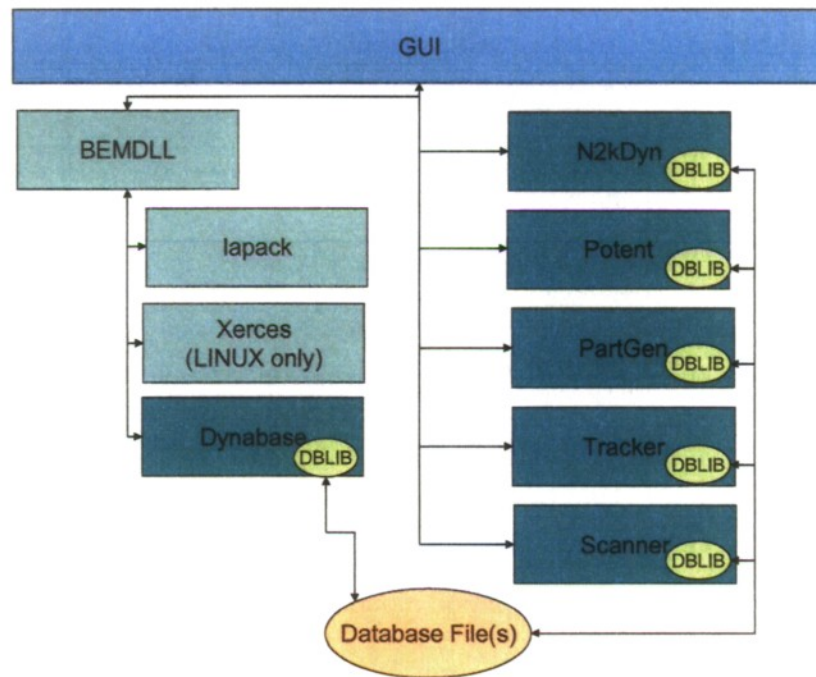


Figure A1. *Nascap-2k 3.2* structure of Main components.

Table A1. Major components of *Nascap-2k* 3.2.

Component	Nature of component
GUI	jar file written in Java
BEMDLL	dynamically linked library written in C++
Lapack	dynamically linked library written in C++
Xerces (LINUX only)	dynamically linked library
DynaBase	dynamically linked library written in Fortran with C++ wrapper
N2kDyn	dynamically linked library written in Fortran with C++ wrapper
Potent	dynamically linked library written in Fortran with C++ wrapper
PartGen	dynamically linked library written in Fortran with C++ wrapper
Tracker	dynamically linked library written in Fortran with C++ wrapper
Scanner	dynamically linked library written in Fortran with C++ wrapper
Database files	Multiple files
DBLib	Library written in Fortran with C low level subroutines

A.1 DBLib

DBLib is the library which provides for all database access in *Nascap-2k*. It does this through three subroutines: `dbfile`, `dbdata`, and `dbinfo`. The input to each of these subroutines is a string, which consists of keywords and arguments that instruct the underlying code what is to be done. The first subroutine, `dbfile`, handles opening and closing of the database. The third subroutine, `dbinfo`, handles creation of, and inquiries about, the data structures in the database. The subroutine `dbdata` handles reading from and writing to the database. If the keyword `HERE` appears in the input string to `dbdata`, the data is written to or read from the specified location in memory provided as an argument to the `HERE` keyword. When `dbdata` is used to read data from the database into memory, the memory location at which it is placed is always returned at the location `idboff(1)`, which is in the common block `/odbbcm/` in the include file `oddbdata.h`.

A.1.1 Buffio

The subroutine `buffio` (in **dynalib**) functions as a wrapper for `dbdata`. It is used for reading up to four gridded items for a single grid at once. Generally, `buffio` is used, rather than `dbdata` directly, for gridded data.

A.1.2 Dbdata

The subroutine `dbdata(String command)` allocates and releases memory, reads and writes from disk files to memory, and initializes data values in memory (data handling). The argument to `dbdata` is a keyword-oriented input string like “read data=pot_20_node grid=2” and output from the subroutine is returned in a common block in the include file `oddbdata.h` as described above.

If there is an error reading data from the database, then the logical flag `ldberr` is set to true, and the error message, `cdber` is set to reflect the error condition. Both of these variables are in common blocks in **odbdata.h**. These conditions are:

- Data type not defined, no action taken
- Data not written, read zeroes instead

This structure works in the following manner. After calling `dbdata` to request data from the database, the calling routine checks the error flag, and then reports that, for the data requested, it was not defined or set to zero if there was a problem.

A.1.3 Dbinfo

The subroutine `dbinfo(String command)` defines and determines the data (data about the data, or metadata). The argument to `dbinfo` is a keyword oriented input string like “DEFINE Data=pot_8_node element_type=Node_8 dimension=scalar value=real set_value=0” and the output from the subroutine is returned in a common block in the include file **odbinfo.h**.

If there is an error defining and determining the data from the database, then the logical flag `ldierr` is set to true, and the error message, `cdierr` is set to reflect the error condition. The variables `ldierr` and `cdierr` are in the common block in **odbinfo.h**. Currently `cdierr` is never set, and `ldierr` is always true.

A.1.4 Dbfile

The subroutine `dbfile(String command)` is an interface subroutine for file handling. It defines the files (file manipulation) and their contents. The argument to `dbfile` is a keyword oriented input string like “Open Prefix=Dmsp STATUS prefix=Data_set_1 PREFIX=Data_set_2” and the output from the subroutine is returned in a common block in the include file **odbfile.h**.

If there is an error during this file handling, then the logical flag `ldferr` is set to true, and the error message, `cdferr` is set to reflect the error condition. The variable `ldferr` is in the common block in **odbfile.h**. The only use is the generation of the message “Could not find anything familiar in input.”

A.1.5 DbLib Errors

Additionally, other errors are returned directly by the subroutines in **DbLib** to report when there are problems in database access or memory management. These error messages include the routine name and the problem encountered. The errors appear in Table A2.

The underlying **MSIO** code generates the error messages given in Table A3. (Note that **MSIO** identifies files by their logical unit numbers, *lun*.)

Table A2. Error messages reported by DbLib subroutines.

Failure to open a database file or database not opened prior to access.
Error closing file(s).
Can't find file(s) associated with the current problem.
Cannot determine type of database file accessed.
Error reading from or writing to the database. This includes when error occurs when data in the database is being overwritten. Also happens when "key" data can't be found in the database.
Error handling "dead space" in the database files.
Reached/exceeded maximum size allowed for any type of data, for example, grids, lists, time names, etc.
Memory manager problems: couldn't allocate/de-allocate memory, tried to de-allocate memory in use, or ran out of buffer space.
Placing several items in same place – pointing to a space allocation error.
Unknown keyword in the command string or command string not in the correct form.
No dynamic memory.
Couldn't find original data from which new data is defined with "same-as".
Name duplication of stored data.
Illegal name for a data type or data type not defined correctly if there is a certain definition expected for the data. This happens when positions are stored, which requires three components, or if the components has known limits, such as the mesh size for a grid, or the component types are expected to be of a certain type, such as integer for the number of surfaces, real for a position, etc.

Table A3. Error messages reported by existing MSIO.

Readms/writms/closms – ms file <i>lun</i> not opened
Openms-cannot open <i>lun</i> = <i>lun</i> , would exceed the max number of ms files= <i>maxmsfiles</i> .
Readms-tried to read key <i>keynumber</i> from file <i>lun</i> which has a max of <i>maxkeys</i> keys. (Only for number key files.)
Readms – <i>lun</i> <i>lun</i> , key <i>keyname</i> , not previously written. Read zeros instead.
Readms – <i>number</i> words written, <i>number</i> words requested.
Fastrw error in readms -- <i>lun</i> = <i>lun</i> , KEY= <i>keyname</i> , expected <i>number</i> words, found <i>number</i> words.
Writms-tried to read key <i>keynumber</i> from file <i>lun</i> which has a max of <i>maxkeys</i> keys. (Only for number key files.)
Writms-index file full for ms file <i>lun</i> .
Writms error on <i>lun</i> <i>lun</i> Entry Key(1) = <i>value</i> [octal] Present file size = <i>filesize</i> [octal] x 512 bytes Proposed addition = <i>bytesneeded</i> [octal] x 512 bytes No Action Taken
In Writms, no length for record, new value of <i>number</i> entered. (Warning only.)
In Writms, attempt to extend record. Key = <i>keyname</i> , new length = <i>number</i> old length = <i>number</i> . This will be treatd as a new record. (Warning only.)
Writms – wrote <i>number</i> words on file <i>lun</i> at key <i>keyname</i> . (Generated only when diagnostics is turned on.)
Fastrw error in writms – <i>lun</i> <i>lun</i> , expected to write <i>number</i> words, wrote <i>number</i> words.
Writms <i>lun</i> = <i>lun</i> key = <i>keyname</i> , number of words = <i>number</i> , No action taken. (Generated when <i>number</i> is <=0.)

A.1.6 DbLib Commands

There are 204 calls in 62 Fortran subroutines to `dbinfo`. There are 298 calls in 91 Fortran subroutines to `dbdata`. There are 82 calls in 30 Fortran subroutines to `dbfile`. There are 183 calls in 34 Fortran subroutines to `buffio`.

The keywords used in the `dbdata`, `dbinfo`, and `dbfile` command strings are given in Table A4.

Table A4. Keywords used in present database commands.

Keyword	Function
dbdata	
OPEN	Allocate space for data in memory. ¹
CLOSE	Deallocate space for data in memory.
READ	Transfer data from disk to memory.
WRITE	Transfer data from memory to disk.
INITIALIZE	Initialize a data type to its "set_value".
HERE	Point to a memory address, use idbloc() to get offset from idata(0). Used to tell the database where something is.
LENGTH	Size in words needed for data in memory. Used when saving Matrix_Elements_## and Bound_Surfs to specify actual length of record.
DATA	Specify a data name.
GRID	Specify a grid number.
TIME	Specify the time stamp.
PREFIX	Specify the file prefix. The value of Scratch is used for storing values in a temporary file.
dbinfo	
DEFINE	Define or redefine a piece(s) of information
INQUIRE	Request information about...(More information below.)
DATA	Specify a data name.
PROBLEM	Only used with INQUIRE. Get the generally useful stuff about the problem.
LIST_DEF	Used with DEFINE and INQUIRE. When used with DEFINE, specifies the length of the specified list. The specified list can then be used as the argument for the LIST_TYPE. Values used are SURFACE_LIST and SURFACE_NODES.
GRID	Used with INQUIRE to read information about the grid and with DEFINE to specify the grid.
SAME_AS	The new data item is the same as a previously defined data item.
DATA_TYPE	Data is SPATIAL, BUFFER, or LIST
DATA_LENGTH	Number of elements.
LIST_TYPE	Name of list definition for this data item. Defined in LIST_DEF command. Values used are SURFACE_LIST and SURFACE_NODES.

¹ All dbdata calls except the following include the OPEN keyword. In Spaini.F there are calls to dbdata with commands to WRITE and WRITE CLOSE for the data POTS and POT_Surf, RHO_Ion_Surf, POTG, and POT_Grid. In pupda2.F and setcnd.F, there are calls to dbdata with CLOSE for data Pot_Bias_Surf.

Table A4. Keywords used in present database commands. (cont.)

dbinfo	
DATA_DIM	Length of each item in surface or spatial grid list. Never used with BUFFER data.
VALUE_TYPE	Type of data: REAL/INTEGER/LOGICAL/OCTAL/ALPHA
ALPHA_LENGTH	Length of ALPHA types in characters. If not specified defaults to 80.
SET_VALUE	Initialization value
OWN_FILE	Save data to a separate file. If argument is LONG_NUMBER # or TWO_WORD #, where # is an integer, the separate file is a number key file with # records.
FILE_SUFFIX	File suffix to use for data saved to separate file.
GRID_DEPEND	TRUE/FALSE
TIME_DEPEND	TRUE/FALSE
LIST_LENGTH	Used to LIST_DEF to define length of list.
OUTSIDE	This grid encloses the other grids. Used in DEFINE GRID=1 command to specify that this is primary grid.
INSIDE	This grid is inside another grid. Used in DEFINE GRID command.
GRID_SIZE	Grid mesh size. Used in DEFINE GRID command.
MESH_RATIO	Ratio of inner to outer mesh units. Used in DEFINE GRID command.
PRIM_RATIO	Ratio of inner to primary mesh units. Used in DEFINE GRID command.
ORIGIN	Used in DEFINE GRID command.
ORIGIN_PRIM	Origin location in primary grid units. Used in DEFINE GRID command.
EDGE_LENGTH	Length_x, length_y, length_z: Number of nodes along the grid edge, including end points. Used in DEFINE GRID command.
dbfile	
OPEN	Assign file prefix to run.
CLOSE	Close file prefix.
EXIT	Close all open files.
PREFIX	Prefix to which command applies.
FILE_TYPE	Always DYNAPAC.
SAVE_FILE	Save file when done. The value FALSE is used with the PREFIX Scratch for temporary storage. This does not work in the present implementation.
DIAGNOSTIC	ON or OFF. Generate diagnostics. While apparently not used at present, this functionality could be useful.

All of the **dbinfo** Inquire commands place information in the common block /odbicm/ in include file **odbinfo.h**. Specifically:

call `dbinfo('Inquire Problem')` returns the number of grids in `idingd` and `xmesh` for the main grid in `rdimsh`.

call `dbinfo('Inquire List_Def=SURFACE_LIST')` returns the number of surfaces in `idilsl`

call `dbinfo('Inquire List_Def=SURFACE_NODES')` returns the number of nodes in `idilsl`

Call `dbinfo("Inquire Grid= #")` returns the information given in Table A5 for the specified grid.

Table A5. Information returned from `dbinfo(Inquire Grid)` commands.

Data	Location
<code>nxyz(3)</code>	<code>idigln(3)</code>
<code>IGrid</code>	<code>idiggd</code>
<code>IParen</code>	<code>idiggo</code>
<code>MsRati</code>	<code>idigmr</code>
<code>M1Rat</code>	<code>idigm1</code>
<code>XmLocl</code>	<code>rdigsz</code>
<code>ParOri(3)</code>	<code>rdigor(3)</code>
<code>PriOri(3)</code>	<code>rdigo1(3)</code>

The Fortran subroutines `dbdata`, `dbfile`, `dbinfo` all have the same structure. They each consist of calls to three subroutines. The first of these defines constants and handles initialization, the second reads the keyword input and sets the appropriate variables and flags, and the third performs the requested actions.

A.1.7 Summary of DBLib Functionality Used

At its root, **DBLib** opens and closes the database, specifies data structures, writes data from memory to disk, reads data from disk to memory, requests space for data in memory, and releases the space when it is no longer needed. It allows for initialization of data to a specific value. Data can either be read from or written to a specific memory location or memory can be handled by **DBLib**.

Data structures are created to allow for automatic association of data items with surface elements, surface nodes, or volume elements of the spatial grid. A grid or surface quantity can be time dependent.

DBLib has special functions to specify the grid structure and the grid parameters.

A.2 DynaBase

DynaBase has the entry points given in Table A6. All of these entry points are called by the **BEMDLL** module, although some are solely for the purpose of passing along the data to/from the user interface.

Table A6. DynaBase entry points.

Entry point	Data used by
CleanHistory	BEMDLL
CloseFiles	Scanner, BEMDLL, GUI
DynaFinish	Not used
DynaInitialize	Used internally, BEMDLL
GetElements	BEMDLL
GctMaterials	BEMDLL
GetNodes	BEMDLL
GetObjectOffset	GUI
GetPotentialsAndFields	GUI
GetPrimaryGridParams	BEMDLL, GUI
GetTimeSteps	BEMDLL, GUI
PutElement	BEMDLL
PutMaterial	BEMDLL
PutNodes	BEMDLL
ReadPotentials	BEMDLL, GUI
ReadSpecies	GUI
SaveHistory	BEMDLL
WriteFields	Not used
WriteMesh	BEMDLL
WritePotentials	BEMDLL
GetGroundingEdges	BEMDLL
GetInitialPotential	BEMDLL

A.3 Data Requests from Nascap-2k Java Interface

The *Nascap-2k* user interface provides the user with data from the database. The **BEMDLL** methods called using JNI are listed in Table A7. These methods in turn call the **DynaBase** entry points given in Table A6. Most of the JNI calls return a Boolean that indicates success.

Table A7. Data requests from user interface.

Java class	JNI method	Corresponding DynaBase entry point/Comment
IonPlumeTableData	GetObjectOffsetFromDatabase	GetObjectOffset
GetParticleSpecies	GetParticleSpecies	ReadSpecies
Mesh	readDynapac	Calls BEMDLL which reads the geometry from the database, constructs the elements and sends it back to the Java, one element at a time.
Mesh	numDynapacElts	Gets information from BEMDLL rather than database
Mesh	closeDynapacFiles	CloseFiles
Mesh	getPrimaryGridParams	GetPrimaryGridParams
Mesh	getElement	Gets information from BEMDLL rather than database
FileUtilities	setDirectory	Calls _chdir
ChargingHistory	getTimeStepTimes	GetTimeSteps
ChargingHistory	getPotentials	GetPotentialsAndFields
ChargingHistory	getFields	GetPotentialsAndFields
ChargingHistory	getCurrents	GetPotentialsAndFields
ChargingHistory	getChargeCurrents	GetPotentialsAndFields
ChargingHistory	getTrackCurrents	GetPotentialsAndFields

A.4 Data

This section describes each of the data items in *Nascap-2k* 3.2. There are three kinds of data: surface, grid, and other. Surface information is information associated with either each surface or each surface node. Arrays of surface or surface node information are always dimensioned by MaxSrf. The surface data items are given in Table A8 and Table A9. The local variable name is that used by the Fortran. If the local variable name is listed, then at least once the data is accessed using the HERE keyword, otherwise, the data is always managed by the memory manager. Four data items are time dependent. Four data items are saved in the Scratch file.

Grid information is associated with each volume element, each node of each volume element, or the 32 interpolants of each volume element. The gridded data items are given in Table A10. Grid data is *always* handled by the memory manager. POTG is time dependent. Five data items are saved in the Scratch file.

There are several data items stored in the database that are neither grid nor surface data. They are listed in Table A11 and described further below.

The grid structure is saved by explicit `dbinfo(Inquire/Define Grid=)` commands.

The BEM matrices are stored separately from the database in the file *prefix*BEM.BEM.

Table A8. Data items associated with surfaces.

Data Item Name	Dimension	Time depen.	In scratch file	Local variable name	Common block	Include file	Data Type
Surf_Elems	4			Nodes	/Srfacs/	surfinfo.h	
Centroids	3			Cntrds	/Srfacs/	surfinfo.h	
Surf_Norms	3			Snorms	/Srfacs/	surfinfo.h	
Surf_Attribs (material, conductor,...)	10			Attrib	/Srfacs	objinfo.h	Integer
Diag_Surfs (diagonal matrix elements for surfaces?)	2			DgSurf	/DagSrf/	none	
POT_Surf	2			Pot		none	Real
POTS	2	yes					Real
CHRG_CURRENT_Surf	1	yes		ChargeCurrents		none	Real
CURRENT_Surf	1	yes		TrackCurrents		none	Real
POT_Bias_Surf	1	yes		Potentials		none	Real
FIX_Surf	2			ISrfFix		none	Alpha:4
Old_Pot_Surf	2						Real
R_Surf	2		yes				Real
U_Surf	2		yes				Real
AU_Surf	2		yes				Real
DINV_Surf	2		yes				Real
RHO_Surf	2						Real
RHO_Ion_Surf	2						Real
RHO2_Surf	2						Real
PLOT_Surf	1						Real

Table A9. Data items associated with surface nodes.

Data Item Name	Dimension	Local variable name	Common block	Include file	Data type
RNODES	3	Rnodes	/Nodcom/	Nodeinfo.h	Real
Node_Surfs	12	Jsurfs	/Nodcom/	Nodeinfo.h	Real
Node_Surf_Weights	12	Snwgts	/Nodcom/	Nodeinfo.h	Real
RNode_Radii	1	RRadii	/Grounding/	none	Real

Table A10. Data items associated with volume elements.

Data item name	Data items per element	In scratch file	Data type
SCRN_Grid	Element centered	yes	Real
EFLD_Grid	Element centered		Real
RHO_GI	Element centered		Real
NeutDens	Element centered		Real
Ion_Plume	Element centered		Real
RHO_Elec	Element centered		Real
RHO_CEX	Element centered		Real
LTBL	Element centered		Integer
POT_Grid	32 Node		Real
Old_Pot_Grid	32 Node		Real
R_Grid	32 Node	yes	Real
U_Grid	32 Node	yes	Real
AU_Grid	32 Node	yes	Real
DINV_Grid	32 Node	yes	Real
RHO_Grid	32 Node		Real
RHO2_Grid	32 Node		Real
RHO_Ion	32 Node		Real
POTG	32 Node (time dependent)		Real
PLOT	32 Node		Real
CUR_Vec{X,Y,Z}	32 Node		Real
NTBL	8 Node		Integer
Nearst	8 Node		Integer

Table A11. Data items not associated with surfaces or volume elements.

Data item name	Size	Local variable name	Common block	Include file
History	100			none
Materials	316		/MatlsC/	ObjSurf.h
Time_Params	MaxTim		/TimeCm/	timeinfo.h
Flux_Table	202		/TabCom/	pginfo.h & psinfo.h
Current_Table	1+NTParm*MxTime		/CurTab/	timeinfo.h
ObjectDimensions	7	Dxyz, Cxyz, CnvFac	/ObjDim/	
Conductor_History	CndHst(2, MaxCnd, MxTime)	Potentials Currents	/CndHst/	none
POT_Conductor	7*MaxCnd+3		/VConds/	none
Matrix_Elements_##	MaxME		/SPSTR2/	specell.h
Bound_Surfs	9999			none
Particles_Record_1	1+4*MAXSPE		/PTinfo/	ptdata.h
Particles_Record_2	10		/PTinf2/	ptdata.h
Particles_#	Essentially unlimited	See below		
Q_Conductors		QCond		

The data item `History` contains 8 pieces of information. They are

Algorithm (32_NODE, 8_NODE, or RESERVED), which is an outdated parameter
 DeBLim
 Debye
 Temp
 Dens
 ObjVel(3).

The data item `Materials` contains the contents of the common block `/MatlsC/`, the number of materials, the names of each material, and the 20 properties associated with each material. `/MatlsC/ NMat, MCode(MaxMat), MatPr(20,MaxMat)`. Saving the information in this manner enforces a 4 character limit on the material name, a limit of `MaxMat` materials, and 20 properties per material.

The data item `Time_Params` contains the contents of common block `/TimeCm/`, which has contents `TimStr, TimRis, TimFal, LTimeUpdate, TimXt2(6)`.

The data item `Flux_Table` contains the contents of common block `/TabCom/`, which has the contents `Ntable, TableI(201)`. `TableI` is an ion flux table that is used to compute sheath weights.

The data item `Current_Table` contains the contents of common block `/CurTab/`, which has the contents `NTime`, `CurTbl(NTParm,MxTime)`. The values in `CurTbl` for a given time are the contents of `ActTim`. The contents of `/ActTim/` are `ITime`, `DelTim`, `TotTim`, `TotCur`, `TotLos`, `TotTrp`, `TotOth`, `LPotSv`, `TimXtr(2)`.

The data item `ObjectDimensions` contains the extent of the object in X, Y, and Z, the center of the object with respect to the grid center, and the scale factor between the Object Toolkit dimensions and the size used. These values are read from the grid definition file.

The data item `Conductor_History` contains conductor potentials and currents. Values are read and written by **DynaBase**. The common block is used to format data for saving to disk and reading from disk.

The data item `Pot_Conductor` contains the contents of common block `/VConds/`, which has the contents `VCond(MaxCnd)`, `CType(MaxCnd)`, `IBFrom(MaxCnd)`, `PotAWA`, `PotMax`, `ICLow`. Saving the data in this way limits the number of conductors to `MaxCnd` and prohibits changing this common block.

The data item `Matrix_Elements_##`, where `##` is an integer, contains the contents of common block `/SPSTR2/` for a special element in grid `##`. It has contents `NCnt`, `LCent(NCntMx)`, `NCorns`, `LCorns(8)`, `VESQ(MAXDIM)`. (`NCntMx=70`, `NNodMx=32+2*NCntMx`, `MAXDIM=2*NNodMx**2`). It has a secondary index of the special element number.

The data item `Bound_Surfs` contains bounding surface information for a special element. The information is created and written out by **N2kDyn**. It is then read by `rsurfs.F` and unpacked into the contents of common blocks `/CNodes/`, `/Surfcs/`, and `/CntCom/`. The contents of these common blocks are `/CNodes/ NCNode`, `RCNode(3,NScsMx)`, `/Surfcs/ NCsurf`, `Jcsurf(4,NScsMx)`, `SCNorm(3,NScsMx)`, `PArea(NScsMx)`, `/CntCom/ Ncnts`, `MatCnt(NCntMx)`, `LTrans`, `LTCnt(1000)`, `TNdCnt(4,2,1000)`.

The data item `Particles_Record_1` contains the beginnings of common block `/PTinfo/`, which contains `NumSpe`, `PChar2(MAXSPE)`, `PMass2(MAXSPE)`, `NPtTot(MAXSPE)`, `NPage(MAXSPE)`. The `NActiv` array, also kept in this common block, is not saved in the database (at least not any more). Saving the data in this way enforces the limit of `MaxSpe` species and prohibits changing this common block.

The data item `Particles_Record_2` contains the contents of the common block `/PTinf2/` for each species. The contents of the common block is `CSpeNm(MAXSPE)`, `PTxtr1(8)`, `Filler`. The last two items are extras for padding. This data record appears to contain only the species names.

The data items `Particles_#`, where `#` is an integer, contains the particle information. The data is stored in pages, with the header of each page being the contents of common block `/PTDesc/`. The common block contains `ISpeci`, `JSpeci(2)`, `PCharg`, `PMass`, `NParts`, `IPage`, `IPType`, `PTxtr2(12)`. The information stored for each particle is that of common block `/ActPrt/`. This common block contains `/ActPrt/ PrtPos(3)`, `PrtVel(3)`, `PrtWgt`, `IPrtSt`, `IPrtGd`, `PrtTim`, `NPrtSt`, `PrtEgy`, `PrtTemperature`, `PTxtr3(7)`. These data items are saved to separate files, one for

each species. Storing the data in this way prohibits changing the structure of these common blocks. `NPage` in `Particles_Record_1` keeps track of how many pages of particles there are. The header requires that different “type” particles must be different species. *Nascap-2k* does insert “PLOT” in the species name of particles for plotting.

`Q_Conductors` appears to never be defined and therefore is not saved correctly.

A.4.1 Summary Comments

The data structures in *Nascap-2k* 3.2 allow for data to be indexed by the items given in Table A12. Data items can be dimensioned to allow for “vector” quantities, such as the three components of the surface normal or the ten “attributes” associated with each surface. Potentials and currents can be time dependent as well as surface, grid, or conductor number dependent. There are a very few common blocks and quantities not associated with volume elements, surface elements, or surface elements that are stored. The fact that entire common blocks are saved as a single data item means that changes in array limits make all existing databases out of date.

Table A12. Quantities that are used as indexes.

Index quantities
Surface element number
Surface node number
Volume element (grid, element number, node/interpolant)
Special element number
Conductor number
Particle species
Particle number within species
Time (a second index)

A.5 Data Storage

MSIO presently can use any of three methods to track the data in the database file: number key, name key, and two-word key. The details of the index structure are responsible for some of the limits on data record and file sizes.

For number key and two-word key files, the key is an integer pointer. For number key and name key files, the index stores the starting location of the record and the length of the record in a single word. For two-word key files, the starting location and length are stored in consecutive words. The largest possible key is one less than the length of the index for number key files and half of one less than the length of the index for two-word key files.

For name key files, the key is a string. For a given file, the key can be any multiple of 4 characters, but the string must be the same length for each record in the file. The keys are stored in the index with the word containing the location and length of the record following the key.

For number key and name key files, the location of the last record must be less than $(2^{16} - 1) \times 128$ words/sector $\times 4$ bytes/word. Thus the maximum file size is about 32 MB. That the data is stored in 128 word sectors was a compromise to maximize file size and record length while minimizing slack space subject to the constraint that record length and location must be stored in a single word.

For number key and name key files, the length of a record is limited to $2^{16} - 1 = 65535$ words. As a grid's worth of information is stored in a single record, this limits the number of nodes for a grid to 16383 (4 pieces of information per grid point.)

For two-word key files, the file size limit is on the order of 2 terabytes, and the length of a record is limited to $2^{32} - 1$, which is greater than 4×10^9 words.

DBLib adds an additional layer on top of **MSIO**. **DBLib** maintains a list of all data items, the file in which each is stored, and the number key used by **MSIO**. (**DBLib** uses only number key **MSIO** files.) **DBLib** also maintains general problem information accessed through `dbinfo`, `DEFINE` and `INQUIRE` commands.

A.6 Searches Contemplated

We would like to be able to obtain time histories of, or values at a specific time (ordered by magnitude) for, potentials and currents (all components) for groups of surface elements where the group is constructed by an arbitrary set of logical operators. These operators would be unions and intersections of surface elements of specified materials, conductors, orientation, sun/shaded condition, and range of values of potential or current component at the final state. Developing the desired list of surfaces given the search criteria might be easiest done in Fortran, C++, or Java. The database could then be queried for the time history of the relevant quantities for the list of surface elements.

APPENDIX B. NEW DATA ITEM NAMES

Each item stored has a unique case-insensitive 16 character (4 words) identifying name key

Time dependent data is stored in a file that uses a 16-character name key and a number key in the following word as an identifier (5 words). (This means that time-dependent data is stored in different file(s) than present, non-time-dependent, data.) The number key gives the time index.

For time-dependent data, the same name key is used for both the time-dependent and present value.

In the underlying MSIO index, the characters are strictly upper case. The characters are converted to upper case by MSIO on reading and writing, so that the rest of the code can be insensitive to case.

We are taking this opportunity to switch to a new set of data item names that provide more information about the data. Data item names start with a few characters that specify the type of data, as in surface, node, grid, particle, material, etc. Each gridded data item has a data item name of no more than 12 characters, an underscore, and three digits that specify the grid. (12characters_###). (Data item names are *not* padded to make sure they are 12 characters long.) The new data item names are given in the tables that follow.

The data stored with data item names R_Surf, U_Surf, AU_Surf, DINV_Surf, R_Grid, U_Grid, AU_Grid, and DINV_Grid will ultimately not be kept in the database. Therefore, we are not giving them new data item names. SCR_N_Grid will be moved from the scratch file to the permanent file. Q_Conductors is presently not saved correctly, so it will be deleted.

Table B1. Data items associated with surfaces.

Old Data Item Name	NewData item name (16 characters max)	Dimension	Time depen.	Local variable name	Common block	Data Type	Incorporated in <i>Nascap-2k</i>
Surf_Elems	Srf_Element	4		Nodes	/Srfacs/		yes
Centroids	Srf_Centroid	3		Cntrds	/Srfacs/		yes
Surf_Norms	Srf_Normal	3		Snorms	/Srfacs/		yes
Surf_Attribs (material, conductor,...)	Srf_Attribute	10		Attrib	/Srfacs	Integer	yes
Diag_Surfs (diagonal matrix elements for surfaces?)	Srf_DiagElement	2		DgSurf	/DagSrf/		yes
POT_Surf	Srf_Potential	2		Pot		Real	yes
POTS	Srf_Potential	2	yes			Real	no
CHRG_CURRENT_Surf	Srf_CurrentCharg	1	yes	ChargeCurrents		Real	yes
CURRENT_Surf	Srf_CurrentTrack	1	yes	TrackCurrents		Real	yes
POT_Bias_Surf	Srf_BiasPot	1	yes	Potentials		Real	no
FIX_Surf	Srf_PotFixed	2		ISrfFix		Alpha:4	yes
Old_Pot_Surf	Srf_PotentialOld	2				Real	yes
R_Surf	R_Surf	2				Real	yes
U_Surf	U_Surf	2				Real	yes
AU_Surf	AU_Surf	2				Real	yes
DINV_Surf	DINV_Surf	2				Real	yes
RHO_Surf	Srf_F0ChargeDen	2				Real	yes
RHO_Ion_Surf	Srf_Charge	2				Real	yes
RHO2_Surf	Srf_F1Derivative	2				Real	yes
PLOT_Surf	Srf_Plot	1				Real	yes

Table B2. Data items associated with surface nodes.

Old data item name	New data item name (16 characters max)	Dimension	Local variable name	Common block	Data type	Incorporated into <i>Nascap-2k</i>
RNODES	Nod_Position	3	Rnodes	/Nodcom/	Real	yes
Node_Surfs	Nod_Surface	12	Jsurfs	/Nodcom/	Real	yes
Node_Surf_Weights	Nod_SrfWeights	12	Snwgts	/Nodcom/	Real	yes
RNode_Radii	Nod_Radii	1	RRadii	/Grounding/	Real	yes

Table B3. Data items associated with volume elements.

Old data item name	New data item name (12 Characters max)	Data items per element	Data type	Incorporated into <i>Nascap-2k</i>
SCRN_Grid	GElm_F1Deriv	Element centered	Real	yes
EFLD_Grid	GElm_EField	Element centered	Real	yes
RHO_GI	GElm_GIs	Element centered	Real	yes
NeutDens	GElm_NeuDen	Element centered	Real	yes
Ion_Plume	GElm_PlmDen	Element centered	Real	yes
RHO_Elec	GElm_ChrgDen	Element centered	Real	yes
RHO_CEX	GElm_CExDen	Element centered	Real	no
LTBL	GElm_LTBL	Element centered	Integer	yes
POT_Grid	GNod_Pot	32 Node	Real	yes
Old_Pot_Grid	GNod_PotOld	32 Node	Real	yes
R_Grid	R_Grid	32 Node	Real	yes
U_Grid	U_Grid	32 Node	Real	yes
AU_Grid	AU_Grid	32 Node	Real	yes
DINV_Grid	DINV_Grid	32 Node	Real	yes
RHO_Grid	GNod_F0PhiF1	32 Node	Real	yes
RHO2_Grid	GNod_F1Deriv	32 Node	Real	yes
RHO_Ion	GNod_Charge	32 Node	Real	yes
POTG	GNod_Pot	32 Node (time dependent)	Real	no
PLOT	GNod_Plot	32 Node	Real	yes
CUR_Vec{X,Y,Z}	GNod_Crnt{X,Y,Z}	32 Node	Real	yes
NTBL	GNod8_NodTbl	8 Node	Integer	yes
Nearst	GNod8_Nearst	8 Node	Integer	yes

Table B4. Data items not associated with surfaces or volume elements.

Old data item name	New data item name (16 characters max)	Time dependent	Local variable name	Common block	Incorporated into <i>Nascap-2k</i>
History	General				yes
Materials	Material_###			/MatlsC/	no
Time_Params	Gen_TimeParms			/TimeCm/	yes
Flux_Table	Gen_FluxTable			/TabCom/	yes
Current_Table	Gen_CurrentTable			/CurTab/	yes
ObjectDimensions	Gen_ObjectSize		Dxyz	/ObjDim/	yes
Conductor_History	Cnd_Potential Cnd_CurrentCharg	yes	Potentials Currents	/CndHst/	no
POT_Conductor	Cnd_Potential Cnd_BCType Cnd_BiasFrom			/VConds/	no
Matrix_Elements_##	MtrxElms_#####			/SPSTR2/	yes
Bound_Surfs	BndSrfcs_#####				yes
Particles_Record_1	Species_##			/PTinfo/	no
Particles_Record_2	Species_##			/PTinf2/	no
Particles_#	Prtcls_S##_**** ## = species num **** = page num				no
None presently	Grid_###			/ActGrd/	yes
None presently	Version				yes

APPENDIX C. N2KDBTEST

We developed a test bed, called **N2kDBTest**, that contains a representative set (not exhaustive) of the database calls made by *Nascap-2k*. Functionality is being verified using **N2kDBTest** before any testing is done with *Nascap-2k*.

N2kDBTest was developed in parallel with, and independently of, the new database and wrapper codes.

N2kDBTest includes examples of all types of `dbdata`, `dbinfo`, `dbfile`, and `buffio` calls presently used. It includes creating, opening, reading, writing, closing. It includes access to data in the main database and in auxiliary files. Appendix A provides a guide to the capabilities needed.

N2kDBTest writes test data. During and after execution, the tester checks that the data is read in correctly, with proper alignment, and without overwriting the beginning or end of the intended space.

N2kDBTest will check that the types of error returns that are currently used productively in *Nascap-2k* are handled and returned correctly.

N2kDBTest was used to verify that premature code exit does not render the database unusable.

N2kDBTest includes both the interim implementation and direct calls to the new database, and is written such that **N2kDBTest** subroutines can be used as examples when converting the interim implementation calls to the final, direct calls.

N2kDBTest tests C++ and Java access to the database (write with Java, read with C++, etc.) and contains sufficient examples to be used as a guide to database use in those languages.

C.1. Details of N2kDBTest

N2kDbTest is a test bed for **N2kDB**. It is a functioning code that uses the database in much the same manner as *Nascap-2k*. It consists of source code, input files (object, project, grid, and text input files) and the anticipated output files—both text and database.

N2kDbTest works with **DBLib**, with **N2kDB**, and in an intermediate manner representative of the intermediate implementation.

The code consists of a Java interface and two dynamically linked libraries. The jar, the two dynamically linked libraries, the input files (`N2kDBTestObject.xml`, `N2kDBTest.grd`, `InFile.txt`, `PGInFile.txt`, and `particles.txt`), and the usual Scheme files should be placed in the working directory. **DynaBase.DLL** should be in the user's path.

Source code for the dynamically linked libraries can be loaded from Source Safe by loading the solution file `$/SpacePhysics/N2K/DBTest.sln` into the same directory as `Nascap2k.sln`.

Source code for the Java is located at `$/SpacePhysics/Nascap2k/N2kDBTest`. The input files are located at `$SpacePhysics/Nascap2k/N2kDBTest/bin`.

The interface is shown in Figure C1.

Primary grid extent is 15 by 17 by 19

Xmesh = -0.6000000238418579

Offset is (0.0, 0.0, 0.0)

There are 3 timesteps

3 ▼

Potentials at timestep 3 are 4.0,6.0,8.0,10.0,12.0,14.0,16.0,18.0,20.0,22.0,
24.0,26.0,28.0,30.0,32.0,34.0,36.0,38.0,40.0,42.0,44.0,46.0,48.0,50.0,52.0,
54.0,56.0,58.0,60.0,62.0,64.0,66.0,68.0,70.0,72.0,74.0,76.0,78.0,80.0,82.
0.84.0,86.0,88.0,90.0,92.0,94.0,96.0,98.0,100.0,102.0,104.0,106.0,108.0,1
10.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,

Name	Mass (kg)	Charge (co...
ELECTRON	9.110E-31	-1.602E-19
HYDROGEN	1.670E-27	1.602E-19

35 ▼

Surface 35 potentials over time are 70.0,71.0,72.0,

Figure C1. N2kDBTest Java interface.

The “Compute” button creates the main database file N2kDBTest.DP. The contents of the file after execution are given in Table C1.

Table C1. Contents of N2kDBTest.DP after execution

Key	Contents
1, 2, 4, 5, 6, 7, 8, 9, 10, 11	Database description information. N2kDB stores this information differently or not at all.
20	Material specification in the format of the MATL common block.
21	Element nodes
22	Element centroids
23	Element normals
24	Element attributes. Material #, Conductor #, four “-1”s (conducting edges), initial potential, and three zeros
25	Node locations referenced to the lower left corner of the grid
26	Node Radii (-1 as undefined)
27	Element potentials and fields, the “potent” part of the test code sets each entry to the entry number plus the time step number.
28	Fix Element array. ASCII data, either “FIX “ or “FLOA” for each element (One extra item, maybe associated with the conductor.)
29	Conductor potentials
30	Potential bias values
31	Contents of ObjDim common block
32	RHO_GI for grid 1, initialized to 1
33	Pot_Grid, for grid 1, first point is grid number, last point is 9999.0, the rest of the points are the previous value + the time step number
34	RHO_GI for grid 2, initialized to 1
35	Pot_Grid, for grid 2, with values the same as grid 1 (except for the grid number)
36	RHO_Surf, 2 x 54, all set to “3.0”
37	100 followed by 101 “-2”s, Flux_Table,
38	15001 entries. First entry is number of iterations. Entry 2 is ‘1.’ Entry 12 is ‘2.’ Entry 22 is ‘3.’ The rest are all zeros, CURRENT_Table
39	2, -1.602e-19, 1.602e-19, 0,0,0,0,0,0,9.1097e-31, 1.67e-27, then zeros, entries 29 and 30 are 1.
40	ELECTRON, HYDROGEN, and blanks

The file N2kDBTest.POTG is also created. Starting with key “20,” it contains the potentials (Pot_Grid) for each timestep. See key 33 above for values. The file N2kDBTest.POTS is also created. Starting with key “20,” it contains the surface potentials and electric fields for each timestep. See key 27 above for values.

Empty CUR and CHG files are also created.

The text output file OutFile.txt is also created.

The “Run PartGen” button creates the files N2kDBTest.PT1 and N2kDBTest.PT2. Each has 4 records. The contents of the records in N2kDBTest.PT1 are given in Table C2. The contents of N2kDBTest.PT2 are the same, except for the substitutions given in Table C3.

Table C2. Contents of N2kDBTest.PT1 file.

Key	Entries	Contents
1	1-20	1, ELEC, TRON, -1.602e-19, 9.1097e-31, 9, 1, 4
1	21-40	7.58333e-1, 3.18333, 1.208333, -1.326109e6, 1.326109e6, 0, 1.0, 0,0,0,0, 10.0, 0.1
1	41-60	7.58333e-1, 3.18333, 1.208333, 0.0,1.875401e6,0.0, 1.0, 0,0,0,0, 10.0, 0.1
1	61-80	7.58333e-1, 3.18333, 1.208333, 1.326109e6, 1.326109e6,0.0, 1.0, 0,0,0,0, 10.0, 0.1
1	81-100	2.416667e-1, 3.18333, 1.208333, -1.326109e6, 1.326109e6, 0, 1.0, 0,0,0,0, 10.0, 0.1
1	101-140	Each set of 20 is identical to the one above except for the velocity in fields 4 through 6.
1	141-200	Each set of 60 is identical to the one above except for the position in fields 1 through 3 and the velocity in fields 4 through 6. There are three particles at each position, each with a different velocity. The velocities are along coordinate axes and at 45° of axes.
2	1-20	1, ELEC, TRON, -1.602e-19, 9.1097e-31, 9, 2, 4
2	21-200	These entries follow the same pattern as for key 1.
3	1-20	1, ELEC, TRON, -1.602e-19, 9.1097e-31, 9, 3, 4
3	21-200	These entries follow the same pattern as for key 1.
4	1-20	1, ELEC, TRON, -1.602e-19, 9.1097e-31, 0, 1, 4
4	21-200	All zeros.

Table C3. Differences between N2kDBTest.PT1 and N2kDBTest.PT2.

N2kDBTest.PT1	N2kDBTest.PT2
ELECTRON	HYDROGEN
-1.602e-19	1.602e-19
9.1097e-31	1.67e-27
1.326109e6	3.097227e4
1.875401e6	4.38014e4

The text output file PGOutFile.txt is also created.

The “Get Results” button reads from an existing database. The first pulldown updates the display of the potentials for all surfaces for a specific timestep. The second pulldown updates the display of the potentials for a specific surface for all timesteps.

The database should also be examined using **N2kDB Tool**.

APPENDIX D. NASCAP-2K TESTING

Functionality of the full code is verified by comparison of standard test cases performed with the old and new databases. Full code tests will be repeated as each section of code is transformed from the wrapped database code to direct calls to the new database.

Initial testing is being performed by running standard test cases with the interim implementation. Any differences between results obtained using the interim implementation and the original code will be understood and, if appropriate, corrected.

Replacement of the interim implementation with the final implementation will be done in work packages of a few subroutines within a single module. It is not necessary to run all test cases after each work package, but at least one of the standard test cases known to exercise the module under modification will be run.

After all the interim implementation code has been replaced, *Nascap-2k* will be modified to take advantage of the new database by relaxing some of the more painful limitations. The full suite of test cases will be run after each such task, as well a new, documented test case that exercises the enhancement. The new test case will be documented and preserved, but need not become part of the standard test suite.

APPENDIX E. REQUIREMENTS VERIFICATION

We have reviewed this document and *Software Requirements Specification for Nascap-2k Database and Memory Manager* simultaneously and verified that the design satisfies the specification. The manner in which each paragraph of the specification is satisfied is described below.

E.1. Introduction

E.1.1. Purpose

N/A

E.1.2. Project Scope

The requirements mentioned in the project scope section are almost all repeated and/or addressed in more detail at later points in the specification document. In this section we only address those issues not repeated. The other issues are discussed in the follow sections.

The use of **MSIO** with a Metadata header and two 32-bit words to specify the record length and location—along with changes in the way species, material, and grid information is stored and the compartmentalization of the code—ensure that **N2kDB** will accommodate the continuing expansion of *Nascap-2k*

The vanilla `ReadDatabase` and `WriteDatabase` commands allow it to be used by any code, including other plasma interactions codes such as *COLISEUM*¹ and *EPIC*.² The `GridData` class can be easily extended to accommodate specialized needs of other codes.

The desire for a wrapper to the new database that understands most of the present string commands is addressed by the Interim implementation discussed in Section 4.1.1.

E.2. Overall Description

E.2.1. Product Perspective

N/A

E.2.2. Product Features

The design of **N2kDB** addresses all of the required features and functions: data storage, data transfer, memory management, data access, data structures.

E.2.3. Operating Environment

Compatibility with all of the *Nascap-2k* operating environments is accomplished by the use of standard C++ and wrapping platform- and compiler-dependent code in `#define` wrappers. During development the code is being tested under the environments presently available at

SAIC, Windows XP, LINUX, and MacOS X UNIX (10.5). The MacOS X Unix is being used to test multiprocessor and 64-bit operations.

E.2.4. Design and Implementation Constraints

No provision is made for pre-existing *Nascap-2k*-generated *DynaPAC* databases.

E.2.5. User Documentation

This design document does not address the requirement to modify the existing *Nascap-2k* manual. This document is the desired documentation.

E.3. System Features

E.3.1. Data Storage Capacity and Format

E.3.1.1. Description and Priority

The **N2kDB** functions (Table 3) allow *Nascap-2k* to specify the data item name and length of any data record. There are some specialized functions for the handling of grid data, species data, material data, etc. that separate the format in which data is stored from the format in which it is used. The primary benefit is to make it easier to increase limits on quantities such as number of grids, without making existing database files obsolete.

While **N2kDB** includes the Memory Manager, `MemoryManager` is a separate singleton class accessed directly or as part of database access.

E.3.1.2. Functional Requirements

E.3.1.2.1 Maximum Database File Size

Storing the location of the start of a record in a 32 bit word means that the largest addressable location is 2^{32} . If data were stored in byte sized chunks, this would allow individual database files up to 4 GB. Using a sector size of 512 bytes allows for files sizes up to 2 TB, which exceeds the 100 GB requirement.

E.3.1.2.2 Maximum Record Size

Storing the length of a record in a 32-bit word means that the maximum record length is 2^{32} words (either 32-bit or 64-bit as specified in the file metadata), which exceeds 10^9 words. This exceeds the 10^7 words requirement.

E.3.1.2.3 Maximum Rows in a Table

The maximum number of records is set separately for each **MSIO** file. Since this value is stored in the MetaData, it can be adjusted. The initial values are set to accommodate the specified number of records. (Section 4.2.3)

E.3.1.2.4 Data Format (ASCII, XML, binary, ...)

The choice to use **MSIO** means that all data is stored in binary format.

The accommodation of 32- and 64-bit words is being addressed by specifying the word size of the database in the **MSIO** metadata and by coding choices. The actual reading and writing of data is data type agnostic. Data types which may be different in 32- and 64-bit environments are being used cautiously. Code whose functionality may differ in the different environments is identified for conditional compilation.

E.3.1.2.5 Data in Single or Multiple Files

The specified files that compose the database are identified in Section 4.1 above.

E.3.1.2.6 File Sharing

The use of a separate file for the matrix elements (*prefix.NSE*), accommodates the desire to share these quantities between projects. Under the present design, under some operating systems two projects are able to access the same file at the same time.

E.3.1.2.7 Need for Standard Access Format

The **N2kDB** functions given in Table 3 are accessed from Fortran, C++, and Java in the same manner. Wrappers in **N2kDB** address the language differences.

E.3.1.2.8 Data Structure

The use of **MSIO** insures that the addition of and increase in the length of new records does *not* make existing **N2kDB** databases obsolete. The changes in the way species, grid, and material information is stored allows for future increases in the number of these quantities without revising the database structure. In addition, the use of a database version allows for conversions from one database version to another and, in the worse case, the rejection of an incompatible database.

The requirement for the size of each record to be stored in the database is addressed by the use of **MSIO**. The `GetLength` function of **N2kDB** provides for easy access to this length.

E.3.1.2.9 Error handling requirements

The approach to error handling provides diagnostic information for normal and debug operations.

E.3.2. Data Transfer Rate

E.3.2.1. Description and Priority

The desired rewrite of portions of the Fortran code to eliminate unnecessary reads and writes is not addressed in this document.

E.3.2.2. Functional Requirements

E.3.2.2.1 Size and Frequency of Reads and Writes

Data reads and writes are handled at the lowest possible level. Sorts to locate keys are done using binary comparisons.

E.3.2.2.2 Allowable Impact on Calculation Time

N2kDBTest is being used to evaluate the speed of reading and writing data in order to identify any problems.

E.3.2.2.3 Error Handling Requirements

The approach to error handling provides diagnostic information for normal and debug operations.

E.3.3. Memory Management

E.3.3.1. Description and Priority

The requirement for a separate Memory Manager is addressed by making `MemoryManager` a separate singleton class accessed directly or as part of database access.

E.3.3.2. Functional Requirements

E.3.3.2.1 Maximum Memory Required

The design of the Memory Manager provides no inherent limits on the amount of memory beyond that of the operating system.

E.3.3.2.2 Allowable Impact on Calculation Time

The impact of memory management operations on calculational time remains to be determined. If the present scheme is too slow, another will be used.

E.3.3.2.3 Error Handling Requirements

The approach to error handling provides diagnostic information for normal and debug operations.

E.3.3.2.4 Data Size Determination

As required, memory requests are for a specific number of words. The requests return a pointer to the location of the allocated memory.

E.3.3.2.5 Multiprocessor Operation

No accommodations are anticipated to be necessary in order for the memory manager to operate smoothly in a multiprocessor environment.

E.3.4. Data Access

E.3.4.1. Description and Priority

The **N2kDB** functions given in Table 3 are accessed from Fortran, C++, and Java in the same manner. Wrappers in **N2kDB** address the language differences.

N2kDB Tool addresses the requirement for a stand-alone database management tool. Section 6.3 gives the design for this tool

E.3.4.2. Functional Requirements

On operating systems that do not allow for read/write access of the same file by multiple processes, simultaneous read access of a database by multiple users is accomplished by opening the database as read only.

Nascap-2k will insure that only a single thread attempts to access the database at one time. Read/Write access to the database from the multiple parts of *Nascap-2k* is accomplished by the first part closing the database before transferring control and the second part opening the database for write access.

Database access synchronization is accomplished by writing the **MSIO** file index each time a database write changes the index.

That the index to the database files is only written when it changes, addresses the requirement that database files will not be changed by read access.

E.3.5. Support of Pre-Existing Databases

It has been decided that *no* database conversion tool will be built. The **MSIO** class can convert an old **MSIO** formatted file into a new **MSIO** formatted file. This feature has proven to be useful during development.

E.3.6. Data Structure

MSIO was chosen as the tool for database file access.

We identified the searches that would be useful to *Nascap-2k* users in Section A.6. The sorting will *not* be done within the database software. The “History” functions of **N2kDB** will simplify access of the time history of potentials and currents for specific surfaces and conductors.

E.4. External Interface Requirements

E.4.1. User Interfaces

N2kDB is a dynamically linked library. Its exposed functions are accessible through *Nascap-2k*. **N2kDB Tool** statically links with its underlying functions.

E.4.2. Hardware Interfaces

N/A

E.4.3. Software Interfaces

The first paragraph and table of Section 6.1 above address the design for a common access format and specifies all of the exposed functions.

E.4.4. Communications Interfaces

N2kDB Tool addresses the desire for a stand alone database management tool.

E.5. Additional Requirements from Section 3.3 of this Document

The additional requirements given in Section 3.3 of this document are addressed as follows.

The desire that the code be written in a cross platform manner is accomplished by the use of standard C++ and wrapping platform- and compiler-dependent code in #define wrappers.

The use of metadata to specify the word length and the storage of data in a data type agnostic fashion makes *Nascap-2k* database files nearly platform independent. Compatibility between Big-endian and little-endian computers could be accomplished through the use of the version number stored at the beginning of the metadata. At this time, we do not plan to address this issue.

Ease of maintenance is being addressed in several fashions. All code and documentation is kept together in the Source Safe database as specified in Section 5.2, coding standards are specified in Section 5.3, the separation of database and memory manager operations into **N2kDB** and the internal structure of **N2kDB** assures encapsulation and generality. The code structures and data storage structures being used will permit examination of data in a debugger to a much greater extent than the present code. Documentation is being written in concert with code development. The source code documentation generator tool **DOxygen** and complete file headers are being used to generate drawings of code structure and lists of properties and methods. As none of the programmers like database jargon, it is not being used.

E.6. Appendix References

1. Fife, J. M. et al., The Development of a Flexible, Usable Plasma Interaction Modeling System, AIAA-2002-4267, Joint Propulsion Conference, Indianapolis, Indiana, 2002.

- 2 I.G. Mikellides, et al., Assessment of spacecraft systems integration using the Electric Propulsion Interactions Code (EPIC), AIAA 02-3667, Joint Propulsion Conference, Indianapolis, IN, July 2002.

APPENDIX F. IMPLEMENTATION PLAN

F.1. Schedule

Nascap-2k 3.2 was released (January 2009) just before the incorporation of the new database and memory management system. The new database and memory management system will be fully functional and tested by the time *Nascap-2k* 4.1 is released at the end of the contract in February 2010.

F.2. Tasks

There are eight steps in the implementation of the new database and memory management system. They are listed in Table F1. The eight steps are further broken down in Table F2. *Nascap-2k* 3.2 does *not* include any of the new database and memory management software and was released shortly before implementation of the new software began. While **N2kDBTest** and **N2kDB Tool** were started before **N2kDB**, their development proceeded in parallel with **N2kDB**.

Table F2 specifies the required skills and level of effort required for each task. In some cases the person with the skills is specified by initial. Myron is only specified for those tasks in which his involvement is essential. Myron and Victoria provide their expertise with design and *Nascap-2k* details to other team members as needed. The level of effort required is a ball park figure primarily for staff planning and determination of the scope of each task. Variances of up to 50% for some tasks may well occur.

Table F1. Implementation steps.

Step	Description
1	Design
2	Write underlying database access software
3	Write N2kDBTest and N2kDB Tool
4	Write N2kDB
5	Release <i>Nascap-2k</i> 3.2
6	Interim implementation
7	Full implementation
8	Test and release <i>Nascap-2k</i> 4.1

Table F2. Tasks to be completed.

Step	Task	Who/Skills	Days or Weeks	
1	Determine if we will use SQLite or MSIO .	Team		Done
1	Resolve issues remaining in Section 9.	Team	2 days	Done
1	Design software. (Write Section 6 of this document)	C++ design	1-2 w	Done
1	Verify design satisfies specification.	V	1 d	Done
1	Specify new data item names.	V&M	1 d	Done
2	Translate MSIO from Fortran to C++ and add additional functionality and unit tests of MSIO . Document MSIO . (MSIO only)	C++	2 w	Done
2	Translate fastio from old C to modern C++ and unit test of fastio . Document fastio . (MSIO only)	C++	1 w	Done
3	Write and document N2kDB Tool . (includes user interface)	C++	2 w	Done
3	Write N2kDBTest and document correct behavior.	V	2 w	Done
4	Write and document N2kDBDataModel class of N2kDB and unit tests of the class.	C++	2 w	Done
4	Write and document Database class of N2kDB and unit tests of the class.	C++	2 w	Done
4	Write and document MemoryManager class and unit tests of the class.	C++	2 w	Done
4	Write and document GridData and History classes of N2kDB and unit tests of the classes.	C++	1-2 w	Done
4	Write and document subroutines needed for interim implementation.	V	1 w	Done
4	Prepare N2kDB documentation document.	V	3 d	Done
5	Release <i>Nascap-2k</i> 3.2.	V&K	2 w	Done
6	Add explicit specification of space for non-gridded data that is presently handled by the memory manager.	V	4 d	
6	Move data initialization to s3set and s3zero .	V or K	2 d	Done
6	Switch to new data item names.	V or K	2 d	Done
6	Eliminate saving of items presently saved to Scratch file.	V	1 d	
6	Implement interim implementation and test off-line. Document testing.	V&K	1 w	Done
6	After interim implementation has been fully tested off-line, switch main line <i>Nascap-2k</i> code to new database using interim implementation.	K	2 d	50%
7	Write Fortran wrapper subroutines.	V	3 d	
7	Convert Fortran portion of <i>Nascap-2k</i> from interim implementation of database to final implementation.	V or K	3 w	
7	Eliminate use of database as extended memory.	V	3 d	
7	Move DynaBase methods used by BEMDLL to BEMDLL .	V or K	2 w	
7	Replace data requests from Java user interface through DynaBase with calls directly to N2kDB .	V or K	2 w	
7	Implement any changes to where general problem information is saved, <i>prefixProject.xml</i> , <i>prefixObject.xml</i> , or database.	V or K	1 w	
8	Test <i>Nascap-2k</i> 4.1 for at least six months of regular use before release.	V&M		

F.3. Revisions Needed to FORTRAN Portion of *Nascap-2k*

The **DBLib** subroutines **dbfile**, **dbdata**, and **dbinfo** will be replaced with direct calls to **N2kDB** functions and possibly FORTRAN wrapper functions that access **N2kDB** and reorganize

the data (such as to place it in correct places within a common block). The changes will be made one data item at a time. Most of the changes will be straightforward. The materials, particles, and conductor information require reformatting of the data.

Presently the material associated with a surface is stored in the attributes array by number. This may need to be rethought.

IPType will be added to the /ActPrt/ common block. The way in which particle locations are to be saved requires not saving extra blank space at the end of the page as is presently done.

There are several items in the common block saved using the POT_Conductor data item name. Only the conductor potential appears to ever be accessed. The rest of the items are for the old time dependence in Potent that is used to represent a sudden applied potential. If this code is ever resurrected, we may also need to save ICLow.

Conductor potentials and currents are saved at the end of the surface arrays. This will no longer be done.

APPENDIX G. OPEN ISSUES

The following open issues remain to be resolved:

Some quantities are presently saved both in *Nascap-2k* database and the *prefixProject.xml* or *prefixObject.xml* files. We need to revisit their interaction.

We need to decide under what circumstances the dead space in the **MSIO** file should be cleared out. It should *not* be done every time the database is closed.