THE MARK III DATA BASE HANDLER

James W. Ryan and Chopo Ma

NASA Goddard Space Flight Center

Bruce R. Schupler

Computer Sciences Corporation

ABSTRACT

As a part of the overall commitment to build the Mark III very long baseline interferometry (VLBI) system, the decision was made to build an integrated software system which would span all aspects of VLBI computer activities from scheduling of the experiments to production of the ultimate geophysical and astrometric parameters. It was decided that underlying this system would be a sophisticated data base handler which would act to tie the programs together. The commercially available data base systems were investigated and found unsuited to our needs. An entirely new data base handler was developed by our group; it was written in Fortran and is currently implemented on the Hewlett-Packard 21MX and the IBM 360/91. The system addresses six major problem areas which plague scientific data analysis: the data base handler should (1) provide for an easily specified method of data interchange among programs, (2) provide for a high level of data integrity, (3) accommodate changing requirements, (4) promote program accountability, (5) provide a single source of program constants, and (6) provide a central point for data archiving.

One of the major areas in which the Mark III VLBI system has been significantly upgraded over the Mark I system is the area of information flow and data handling. The following discussion presents the goals we set for ourselves in developing an information system for the Mark III and how well we achieved those goals.

DATA HANDLING PROBLEMS

VLBI data processing is quite complex and involves no fewer than four major software systems and numerous small programs between the acquisition of data and the generation of geophysical results. In the Mark I system, the interchange of data among the programs was handled using formatted cards or card-image tape files. In some cases, the output of one program was not compatible with the input of the next and required an intermediate step to transform the data. The complications inherent in this arrangement prompted the first attribute which our new data handling system would have: it should provide a common, easily specified means of data interchange among all the programs of the VLBI software system.

VLBI observations which can be used for astrometric measurements will be of interest for some time. It is not inconceivable that observations made now will be of interest 100 years hence; astrometric measurements made several hundred years ago are still reprocessed in present studies. Some of the astronomical phenomena we observe have periods of many thousand years and there is no substitute for a long time base. Moreover, the very complexity of VLBI processing requires the storing of many types of correlative data for every astrometric observation. These requirements necessitated the second attribute of our new data handling system: it should provide for an extremely high degree of data integrity. Not only should the system maintain without error the numeric values of the data, but it should also provide a scheme for retaining the identity of the data elements.

In our formatted card data handling system, addition of new data elements was quite difficult. Usually, an attempt was made to put the new data in some unused columns in the existing format. When the unused columns were exhausted, a partial solution was to replace old data which were no longer of interest. Ultimately, the interpretation of special flags was required to allow proper decoding of the information on data cards. Backward compatibility was often impossible, and each time a new data element was added many of the programs required changes. Experience with these problems defined the third attribute of our new data system: it should easily accommodate changing data requirements. It should be possible to add new data elements without destroying backward compatibility and without requiring program modification simply because the new data exist.

The Mark I data handling system had no provision for tracking the status of programs which produced the data stored in the system. At times, results could not be reproduced because it was impossible to know what versions of the programs generated them. To avoid this difficulty, the fourth attribute of the new data system was specified: it should provide a simple method for programs to add data into the system to log their status.

Another feature of the old system was that each program contained its own set of program constants internally; e.g., the speed of light and the locations of the observatories. It was nearly impossible to trace what constants had been used to generate a given result. Moreover, when a new value of a constant was adopted each program using that constant needed modification. The system was cumbersome and prone to errors. Consequently, the fifth attribute was suggested: the system should maintain a core of important constants which all programs would be required to use.

Finally, the old system provided no uniform method for archiving the data. Various parts of the data were stored on cards or in files maintained by individuals. The cards and files were archived by the individuals in whatever way they chose or perhaps not archived at all. This proliferation and absence of control gave rise to the final attribute of our new system: it should consist of a single set of well-defined files where all data of long-term interest reside and should have a well-defined, nearly automatic scheme to archive these files.

DATA BASE HANDLER SELECTION RATIONALE

The six areas summarized in figure 1 provided a set of specifications for the data handling scheme we desired. The first conclusion we drew from reviewing these specifications was that we needed a data base structure and a handler to manipulate it. In the formatted card scheme, individual users read from and wrote into the data files directly; there was no interface between the users and the data, hence no control. Putting a program between the users and the data immediately generated two further requirements: compatibility with all computers which predictably could be used for VLBI data processing and ability to automatically transfer data created on one of the machines to any other. At the time the goals were specified, a UNIVAC 1108, a CDC 3300, or IBM 360/91, and a Hewlett-Packard 21MX were used in VLBI processing. The common features of these machines and their operating systems dictated that the data base handler must operate on any machine which supported Fortran and sequential files. No such system existed. The only data base available on the 21MX was Hewlett-Packard's IMAGE system, which was not transferrable to the other computers. Thus, we were left with no option but to write our own system.

1. DATA INTERCHANGE BETWEEN PROGRAMS
2. DATA INTEGRITY
3. ACCOMMODATION OF CHANGING REQUIREMENTS
4. PROGRAM ACCOUNTABILITY
5. PROGRAM CONSTANTS
6. DATA ARCHIVING

Figure 1. Data management problems.

The second conclusion was that we did not need the full hierarchical and associative structure of a complete data base system. VLBI data are largely organized in a sequential manner based on time of observation. Data not associated with an observation time are minimal; e.g., the catalog of VLBI observatories. The absence of a requirement for tree structures, daisy chains, and other data associations greatly simplified our task.

STRUCTURE OF THE VLBI DATA BASE HANDLER

The easiest way to explain how the VLBI data base handler achieves its goals is to describe it in some detail. In fact, this explanation will be considerably more detailed than is necessary to use the system and will contain details which are unknown to most of our users.

The system consists of two distinct parts: one, a set of files existing on disk packs and tapes; and two, a set of utility subroutines which allow users to access the information in these files. Users never directly read or write the files and need not know the details of how the data are formatted in the files. To the user, the storage medium is "format free." A user does need to know something about the sequencing of his data in the files but nothing about data in which he has no interest.

Figure 2 shows how a file is structured. Each data base file consists of four major subsets of records:

1. The file begins with the identification records. These consist of seven records of fixed format which contain information such as the data base name, update version number, creation date, etc.

2. Next are an indefinite number of sets of history records. Each time the data base file is updated the user must supply one or more lines of text explaining who he is and what he is doing. These lines are then incorporated into a new set of history records which is added to the output file after the existing history records. The new history records also contain the date/time the update was created, the machine type and facility on which the update was made, and the version of the data base handler in use.

3. The third subset consists of the table of contents records. The VLBI data base handler supports up to 99 different types of data records. Consequently there can be as many as 99 tables of contents. Each type of data record that exists in a file has a corresponding table of contents. The existence of different data record types allows users to segregate data for convenience in processing and efficient use of storage capacity.

4. The final subset of records are the data records which contain the user information. Up to the physical limits of the storage medium, i.e., the size of the largest disk file, there can be any number of occurrences of each type of data record with the different types of data records interleaved in any fashion.

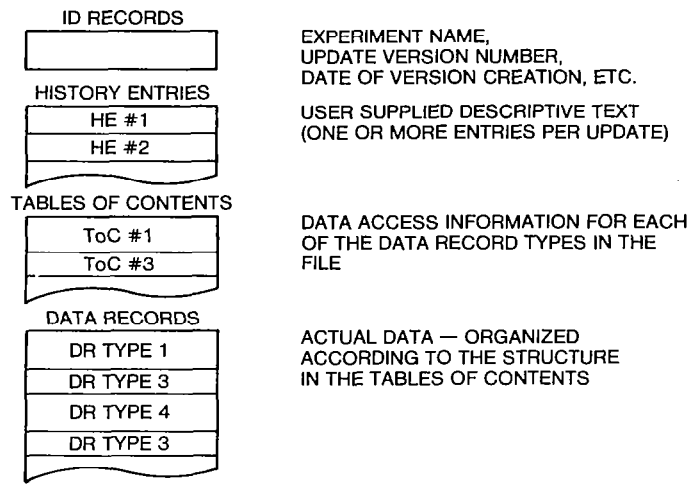

Figure 2. File structure.

TABLE OF CONTENTS ORGANIZATION

The information in the data records is organized into arrays which are accessed by means of 8-character ASCII codes called LCODE's. The correspondence between the LCODE passed by the user and the actual position of the data in the data record is defined in the table of contents. Figure 3 shows the structure of a typical table of contents. It is a matrix of ASCII and integer data in which each row of the matrix contains information pertaining to one array in the data record. A row contains the 8-character LCODE, the version number of the update when the array was inserted or last changed, three dimensions which define the structure of the array, and 32 characters of user-provided text description which identify the array contents. Three types of arrays can be defined – real numbers, integers, and ASCII characters. When the user requests an array he specifies the type desired. The data base handler finds the actual physical location for a particular array in the data record by locating the correct row in the table of contents and using the array types and dimensions of all the preceding arrays in the table of contents.

The version number and test descriptions in the table of contents are features which are essential to achieving two of the goals set for the system. The version number allows the user to know precisely when an array was entered into the system or last modified. The text descriptions play a key part in maintaining the identity of the data elements.

In the example shown in figure 3, each of the three types of arrays is shown. "VLIGHT" is the 8-character code for an array containing the real number which is the speed of light in meters/second. Note that an array containing a single number has dimensions (1,1,1). "NUMBSTAR," which also has dimensions (1,1,1), is an integer which specifies the number of radio sources in the source catalog of the data base file. "STRNAMES," which has dimensions (4,6,1), is list of 6 star names, each of which is 8 characters (4 words) long. It is part of the radio source catalog.

	ACCESS CODE	VERSION NUMBER	DIMENSION 1	2	3	ARRAY DESCRIPTION
REAL ARRAYS	VLIGHT	1	1	1	1	SPEED OF LIGHT (M/S)
	⋮	⋮	⋮	⋮	⋮	⋮
INTEGER ARRAYS	NUMBSTAR	2	1	1	1	NUMBER OF STARS IN CATALOG
	⋮	⋮	⋮	⋮	⋮	⋮
TEXT ARRAYS	STARNAMS	1	4	6	1	CATALOG OF STAR NAMES
	⋮	⋮	⋮	⋮	⋮	⋮

Figure 3. Organization of a table of contents.

It should be emphasized that new tables of contents may be introduced and existing tables of contents may be modified at any update of a data base file. It is not necessary or desirable to define all information and allocate all space at file creation.

DATE BASE HANDLER USER INTERFACE

Users interface with the data base handler by calling the necessary subroutines from their programs. Figure 4 is a list of the various utilities and a short description of their functions. The list is generally in the order of use.

1.	**IRWIN**	— OPEN THE SYSTEM
2.	**PHIST**	— STORE HISTORY ENTRIES
	GHIST	— RETRIEVE HISTORY ENTRIES
3.	**ASK**	— RETRIEVE TABLE OF CONTENTS INFORMATION
4.	**ADD**	— ADDITION TO TABLES OF CONTENTS
	DEL	— DELETION FROM TABLES OF CONTENTS
5.	**OPREC**	— CREATE NEW DATA RECORD
	MVREC	— MOVE TO EXISTING DATA RECORD
	WRIDR	— WRITE DATA RECORD
6.	**GET**	— RETRIEVE INFORMATION FROM DATA RECORD
	PUT	— PLACE INFORMATION INTO DATA RECORD
7.	**FINIS**	— CLOSE THE SYSTEM

Figure 4. Data base user interface.

The system operates in three modes: (1) a "read-only" mode which is by far the most common mode used and in which the user only extracts information from the data base files, (2) an "update" mode in which the user adds new data or extracts and modifies existing data, and (3) a "create" mode in which an entirely new data base is created. In our application, the create mode is rarely used.

The user initializes the data base handler by calling IRWIN. In the call, he specifies the name of the input data base file, the mode (i.e., read-only, update, or create), and the unit number of a terminal or printer where error messages can be sent. IRWIN returns a flag indicating whether the system has successfully opened the file and is ready to proceed. Most other data base utilities also return a flag showing whether the desired function has been properly executed.

The next calls deal with history entries. GHIST returns to the user the history information from the input file. PHIST stores new history records in the output file. In create and update modes, the user must provide at least one history entry; otherwise, the system will not allow the user to continue forming the output file.

At this point in the update and create modes, the user must deal with the tables of contents through the routines ADDA, ADDI, ADDR, DELA, DELI, and DELR. The three ADD routines allow the user to tell the system to provide array space for text, integer, or real arrays, respectively. In the call to the ADD routines, the user specifies the data record type, the access code (LCODE) for the array, the dimensions, and array text description. If the data record type does not yet exist, the necessary table of contents is created. If an array already exists in the input file and the user wishes to modify some of the data stored in that array, the array must be added through one of the ADD routines as if it did not exist. This requirement insures that the version counter correctly tracks the array's last modification. The DEL routines allow the user to delete arrays which exist in the input file. Once an array specification has been deleted from the table of contents, all occurrences of that array in the data records are automatically deleted from the output file. Through the ASK utility, the user can also query the system about the tables of contents of the input file. The user can ask whether an array corresponding to a specific LCODE exists in the file and what its dimensions and text description are.

In order to store data in or retrieve data from a data record, the user must make that record active in the system; i.e., bring the record to a buffer in memory. A record can be made active in one of two ways: by calling the utility MVREC, which activates a record, either by type or sequence, through reading it from the input file; or by calling OPREC, which creates an empty record. Once a record is active, the actual data accesses can occur. Arrays are retrieved and stored by using the utilities GETA, GETI, GETR, PUTA, PUTI, and PUTR. The three GET routines retieve text, integer, and real arrays, respectively; the three PUT routines store arrays in an identical fashion. In the calls to the PUT and GET routines, the user specifies the desired array by passing the appropriate LCODE and a buffer into which the data are returned in the case of a GET and out of which the data are passed in the case of a PUT. The user must also pass the array dimensions in order to insure a correct linkage between the user program and the data base handler system. The dimensions passed to the GET routines can be smaller than those specified for the array in the table of contents; the system will only return as much as specified by the passed dimensions.

In the update and create modes, the user must specifically dispose of the active record when he has finished by calling WRIDR or DELDR. WRIDR causes the active record to be written to the output file. DELDR simply deletes it. By using DELDR, records which exist in the input file but are unwanted in the output file can be eliminated.

After calling WRIDR or DELDR, the user can make another record active by calling MVREC or OPREC. In a typical application, most of the interaction with the data base system involves a large loop beginning with a call to MVREC or OPREC to make a record active, several calls to GET and PUT for actual data access, and ending with a call to WRIDR to output the active record. In this fashion, the user can move through the records of an entire data base file.

When the user is finished with a data base file, he terminates the system by calling the utility FINIS. In the update mode, if the user has moved through only a portion of the input file when FINIS is called, the remainder of the input file is then copied into the output file. This copying is provided as the default to insure that data are not inadvertently lost in updating.

In general, when the data base handler detects some unexpected condition, it sets an error flag and returns control to the user. Such an error condition occurs, for example, when the user attempts to GET an array using an LCODE which does not exist in the table of contents. However, there are abnormal conditions which the data base handler detects which can only occur if some internal condition is completely incorrect or if the user has attempted to misuse the system. A typical example of the latter is an attempt to store data using one of the PUT routines after initially specifying read-only mode in calling IRWIN. In these cases, the system writes a error message in plain English on the device specified in the call to IRWIN, closes the files, and terminates the user program.

SUMMARY - PROBLEMS AND THEIR SOLUTIONS

Figure 5 summarizes the six problems which our data handling system was to alleviate and the solutions to these problems. In review, the requirements were:

1. The system should provide a common, easily specified means of data interchange among all the programs of the VLBI software system. This goal has been accomplished. If two users wish to pass data through a data base file (one generating the data, the other using them), they need only agree on the data record type in which the data are to be stored, the array type and dimensions for holding them, and an 8-character LCODE to access them. Once the data are in the system, any user can access them.

2. The system should provide for data integrity; it should not only preserve the numeric values of the data but also retain the identity of the data elements. The system satisfies this requirement in at least two ways. Associated with each array in a data base file is a 32-character text description. These descriptions if used conscientiously clearly define the data elements. In addition, the history records give each user an opportunity to make a permanent log in the output file of precisely what he was doing when he updated the data base file.

3. The system should accommodate changing requirements. It automatically preserves backward compatibility and does not require programs to be modified simply because new data have been added. With the data base handler, users are isolated from data files and need not know precisely what data are in them or how the files are formatted. New data can be added and a user who is not interested in those data need never know they exist.

1. DATA INTERCHANGE	THE DATA BASE FILES PROVIDE A COMMON, EASILY SPECIFIABLE LINK BETWEEN PROGRAMS. USED ON MANY MACHINES.
2. DATA INTEGRITY	USER NEVER DIRECTLY WRITES DATA FILES. DATA SET MODIFICATIONS CAUSES A NEW FILE TO BE CREATED AND THE OLD FILE IS PRESERVED.
3. ACCOMMODATION OF CHANGING REQUIREMENTS	USERS CAN ADD NEW DATA WITH NO IMPACT ON ON EXISTING DATA. DATA RECORD AND FILE SIZE EXTENSIBILITY LIMITED ONLY BY DISK CAPACITY.
4. PROGRAM ACCOUNTABILITY	USERS MUST ADD HISTORY RECORDS WHEN UPDATING. PROGRAMS CAN AUTOMATICALLY ADD TEXT INFORMATION ON PROGRAM VERSION NUMBER, LAST COMPILATION DATE
5. PROGRAM CONSTANTS	ALL CONSTANTS CAN BE STORED IN THE DATA BASE WHEN CREATED AND ALL PROGRAMS CAN BE REQUIRED TO GET CONSTANTS FROM THE DATA BASE.
6. DATA ARCHIVING	WELL DEFINED INTERNAL FORMAT; DATA DESCRIPTIONS CONTAINED IN TABLES OF CONTENTS. ALL DATA RESIDES IN A SINGLE SET OF WELL CONTROLLED FILES WHICH CAN BE ARCHIVED ON TAPE IN A NEARLY AUTOMATIC FASHION.

Figure 5. Problems and solutions.

4. The system should provide a simple method for programs to log their status. In our application, all data base files contain a data record of type 1. Each file contains only one data record of this type; it is called the "header record" and is the first data record in the file. This record contains all information which is needed on a once-per-file basis; e.g., the core of important constants and the radio source and VLBI observatory catalogs. The various programs of our system store text messages in ASCII arrays in the header record to log their status. These messages contain information such as the load module number of the program and the versions and compilation dates of all the major subprograms. By looking at these messages, we can know what versions of the programs created the output data, and by maintaining copies of obsolete program versions, we can reproduce old results if desired.

5. The system should maintain a core of important constants which programs would be required to use. In our application, all data base files are ultimately derived by updating a single existing file, the skeleton data base file which contains nothing more than the core of important constants. Since all other files result from some sequence of updates to the skeleton file, they retain the core of constants.

6. The system should provide a well-defined, nearly automatic method of data archiving. In this system, all data of permanent interest reside in data base files; therefore, the problem of individuals being responsible for archiving the data they generate is eliminated. There is simply one common pool of data. Since the system obviously produces many files – one might even say it proliferates files – a data base file cataloging system has been instituted. Conceptually, it sits above everything which has been described earlier. The catalog logs the existence of every data base file created, its relationship to other data base files, the disk device on which it resides, and whether it has been archived. One individual is responsible for backing up the files on archive tapes and clearing the disks of files which are no longer in active use. The disk-to-tape and tape-to-disk routines are functions in the catalog system and are used for archiving and restoring and for data transmission between machines.

DIFFICULTIES WITH THE USE OF THE VLBI DATA BASE HANDLER

The data base handler which has been implemented meets all the original specifications. However, the system is not without drawbacks. The following is a list of the principal difficulties:

1. Data access requires much more computer time than in the formatted card scheme. On the IBM 360/91, there is no problem since the time involved is negligible with either scheme. On the HP 21MX, where most processing is done interactively and where increased time means some individual must sit waiting at a terminal, the problem does exist. In a typical minicomputer application, a user accessing nearly all the information in a 200-observation data base file may wait 3 minutes; whereas with formatted cards, he might wait 20 to 30 seconds. A faster CPU would reduce the delay.

2. Formatted cards are ideally suited for sorting. In fact, sorting was one of the reasons why computer cards were invented. Data base files are not easily sorted, and we do not have general purpose data base sorting routines. We have written one data base sorting program to do a very specific kind of sorting, and with minor modifications, it could be made to do other types of sorting.

3. Cards formatted identically can easily be concatenated, usually by simply stacking two card decks one behind the other. Concatenation of data base files is not so simple. Because of the organization of the header record it is not possible to write a single program to merge two arbitrary VLBI data base files. In order to merge two files, one must, in general, make use of the actual sense of the information contained in the files. No mechanical scheme simply involving manipulations of data record types, access codes, and dimensions will work. To date, we have skirted this problem by merging the information from separate data base files in the user programs at the time the information is processed. We are, however, developing a program for concatenating a limited class of data base files.

CONCLUSION

In summary, we are quite satisfied with the performance of our data base handler as applied to our VLBI problem. It has some small drawbacks, but we have been able to work around them, and it is infinitely better than the system of formatted cards which it replaced. In fact, had we continued using formatted cards, it is likely that our entire software system would have collapsed around us.