NASA/CR—2005–213524

DB90–A Fortran Callable Relational Database Routine for Scientific and Engineering Computer Programs

Gregory A. Wrenn
Swales Aerospace, Hampton, Virginia

February 2005

NASA/CR—2005–213524

DB90–A Fortran Callable Relational Database Routine for Scientific and Engineering Computer Programs

Gregory A. Wrenn
Swales Aerospace, Hampton, Virginia

National Aeronautics and
Space Administration

NASA LaRC
Hampton, Virginia 23681

Prepared for NASA LaRC
under Contract NAS1-00135

February 2005

Summary

This report describes a database routine called DB90 which is intended for use with scientific and engineering computer programs. The software is written in the Fortran 90/95 programming language standard with file input and output routines written in the C programming language. These routines should be completely portable to any computing platform and operating system that has Fortran 90/95 and C compilers. DB90 allows a program to supply relation names and up to 5 integer key values to uniquely identify each record of each relation. This permits the user to select specific records or retrieve data in any desired order.

Introduction

Often when writing complex engineering computer programs, large amounts of data need to be organized and stored in a relational format such that the data can be retrieved in a different order than it was created. A database is often useful in this situation by allowing the user to specify relation names and identifying keys associated with each record. These names and keys are then used to retrieve data in any order desired.

The DB90 database provides the user with the capability to define relations, which are named data records of an explicit format. Each relation has a unique name, and is defined as a list of variable names with their associated types and lengths. Up to 5 key names can be used with each relation, which are used to group and identify the records written to the database. Only integer key values are supported in DB90 to simplify the database structure and to avoid problems with testing and matching character strings due to string length, and real numbers due to round off and precision considerations.

A Word About Fortran Data Storage

For a given relation, each row of data has exactly the same length and structure and must be defined in the calling program as a contiguous region of memory. This is accomplished in a Fortran program by defining a row of data as either a structure or a common block. As an example, assume a program wants a relation which contains a real matrix and some integer values.

```
real :: rmat
integer :: irow, icol
common /data/ rmat(3,5), irow, icol
```

or

```
type matrixdef
  real :: rmat(3,5)
  integer :: irow
  integer :: icol
end type matrixdef

type (matrixdef) :: matrix
```

Both of these examples define a contiguous region of memory with the named variables. Care must be taken to define a row of data such that the longer data types are defined first. The reason for this is that most Fortran compilers pad common blocks and structures to ensure that each variable starts on a 4 or 8 byte boundary. Hence, each relation should be defined with double precision real and complex variables listed first, followed by single precision real and complex variables, integer variables and logical variables. Character strings should either be assigned to their own relations, be defined with lengths that are a multiple of 4 or 8 bytes or placed at the end of a common block or structure.

In this example, the real matrix rmat has fixed dimensions of 3 by 5. Both common blocks and structures require all dimensions to be fixed at the time the program is compiled. If a programmer wants to first calculate the dimensions of arrays and matrices during program execution and then use those dimensions to define a database relation, a type of integer pointer can be used in place of the array or matrix name. The same examples above can be rewritten as:

```
integer, parameter :: LONG=kind(1d0)
real, allocatable :: rmat(:,:)

integer(LONG) :: rmatptr
integer :: irow, icol

common /data/ rmatptr, irow, icol
```

or

```
type matrixdef
  integer(LONG) :: rmatptr
  integer :: irow
  integer :: icol
end type matrixdef

type (matrixdef) :: matrix
```

The integer rmatptr replaces the actual matrix rmat in the common block or structure. Before the common block or structure is written to or read from the database, the memory address of the actual matrix must be assigned to the rmatptr variable. This is accomplished with the function DBADR.

```
allocate(rmat(3,5))

rmatptr = DBADR(rmat)
```

if the variable is in a common block or

```
matrix%rmatptr = DBADR(rmat)
```

if the variable is in a structure.

The routine DBADR must be used with long integer variables instead of using the Fortran 90/95 definition of pointers, because there is no standard internal representation of pointers. Different compilers and computers allocate memory for pointers in their own way, and the DB90 routines cannot know how to interpret them.

Several of the database subroutines must pass data between the user and the database. Like with most Fortran subroutines, entire arrays of data are passed by using the array's name. This is the equivalent of passing the starting location of the contiguous region of memory that defines an array, or in this case a data record. Rows of a relation are passed to the database subroutines in a similar manner, but the exact method depends on whether a common block or structure is used to define the relation. If a common block is used, then the first variable of the common block is passed to the subroutine that performs data transfer. If a structure defines the data record, then the structure name is passed to the database subroutine. The database routines use the memory location as the starting address of the entire data record, not just the first variable of the data record.

Typical Usage

All coding examples presented in this paper use Fortran 90/95 syntax. It should be possible to use Fortran 77 syntax for the user's program and still access most features of the DB90 program, but this has not been tested thoroughly, and, therefore, results cannot be guaranteed. The DB90 routines are written in Fortran 90/95 and must be compiled with a Fortran 90/95 compiler.

The first step in defining and using a database is to create the database file. The user must first call DBCREATE to create a new file and initialize the database structure. This creates a new, empty database file in the current directory.

```
call DBCREATE ('dbname')
```

If the specified filename already exists, it will be deleted and the new empty database file will take its place.

To define a relation in the database, the routine DCADDR is called. This will add an empty relation definition, which includes the relation name, key names, variable names, variable types and variable lengths. New relations may be added to the database at any time. For the example described earlier:

```
character(len=8) :: kname(5)
character(len=12) :: vname(100)
integer :: vtype(100)
integer :: vlength(100)
integer :: nk
integer :: nv

kname(1) = 'mat'
nk = 1

vname(1) = 'rmatrix'
vname(2) = 'row'
vname(3) = 'column'

vtype(1) = 1
vtype(2) = 0
vtype(3) = 0

vlength(1) = 3 * 5
vlength(2) = 1
vlength(3) = 1

nv = 3

call DCADDR ('table', kname, vname, vtype, vlength, nk, nv)
```

This will add a definition for a relation named “table” to the database with a single key named “mat” and 3 variables named “rmatrix”, “row”, and “column”. One item to note in this example is the type parameter for the “rmatrix” variable. In the above example, a value of 1 is used for vtype(1). This identifies rmatrix as a real data type. If the pointer definition is used from the second example above, then this type would be 11 instead. See the definition of DCADDR in Appendix I for a more complete explanation of each argument to this subroutine. Each additional relation is defined in this same way, and relations can be added at any time.

If the user needs to test to see if a relation has been previously defined, subroutine DBCHECK should be called.

```
integer :: rows, rowlen

call DBCHECK ('table', rows, rowlen)
```

In this example, if the relation named “table” has already been defined in the database the variable “rowlen” is returned with the number of bytes required for each data record and the variable “rows” is returned with the number of data records already written to the database for this relation. If the relation has not been defined yet, the row length is returned as zero.

Now that the relation “table” has been defined, data rows (or records) can be written to the database. To add a new row of data, the user calls subroutine DBADD.

```
call DBADD ('table', 'mat', 42, ' ', 0, ' ', 0, ' ', 0, ' ', 0, rmat)
```

or

```
call DBADD ('table', 'mat', 42, ' ', 0, ' ', 0, ' ', 0, ' ', 0, rmatptr)
```

for the case of a relation defined using a common block, or

```
call DBADD ('table', 'mat', 42, ' ', 0, ' ', 0, ' ', 0, ' ', 0, matrix)
```

for the case of a relation defined using a structure named “matrix”. In this example, the key named “mat” was used with a value of 42. Notice that information for all 5 possible keys must be present, even when fewer or none are used. Additional rows of data can be written by calling DBADD multiple times, and different key values may be used to identify or group the rows for future reference. For the remainder of these examples, only the structure form of the relation data will be used.

Once relations have been defined and data rows have been written to the database, it would be desirable to retrieve data in whatever order the user wishes. Data is retrieved by first identifying the relation to get data from, along with any keys and their values that are needed to specify a subset of all rows of data contained in the database for this relation. Calling subroutine DBFIND selects the relation from which to read data.

```
call DBFIND (‘table’, ‘ ‘, ‘  ‘, 0, ‘ ‘,‘ ‘, ‘  ‘, 0, ‘ ‘,‘ ‘, ‘  ‘, 0, ‘ ‘,‘ ‘, ‘  ‘, 0, ‘ ‘,‘ ‘, ‘  ‘, 0)
```

Since no key names and values were supplied with this call to DBFIND, all rows of data for this relation are available for retrieval. If DBFIND is called as such:

```
call DBFIND (‘table’, ‘mat‘, ‘eq‘, 42, ‘ ‘,‘ ‘, ‘  ‘, 0, ‘ ‘,‘ ‘, ‘  ‘, 0, ‘ ‘,‘ ‘, ‘  ‘, 0, ‘ ‘,‘ ‘, ‘  ‘, 0)
```

then only data rows having the key named “mat” with a value equal to 42 would be available for retrieval. Once again, notice that information for all 5 possible keys must be present, even when fewer or none are used. Refer to the definition of DBFIND in Appendix I for a complete list of all possible key operations.

Now that a relation name and a subset of data rows has been identified using DBFIND, the actual data rows can be retrieved. The user should first establish how many data rows met the DBFIND search criteria by calling DBINFO.

```
integer :: nrows, irow, jrow
call DBINFO (nrows, irow,jrow)
```

This routine returns in the variable “nrows” the number of rows available for retrieval. If this number is zero, then no rows matched the search criteria and no attempt should be made to read any data rows. If the number of data rows is greater than zero, calls to subroutine DBGET are made to actually read the data.

```
call DBGET (matrix)
```

This call to DBGET retrieves the first row of data matching the criteria from the call to DBFIND and stores it in the structure named matrix. The next matching row of data is retrieved with another call to DBGET. A typical code fragment used to read all of the matching rows would be:

```
do i = 1,nrows
  call DBGET (matrix)
  .
  perform some operations on the data contained in the “matrix” structure
  .
end do
```

It may be necessary to change data within a data row after it was written to the database. This can be accomplished in one of two ways. If you first read a row of data using DBGET, then subsequently calling DBPUT will overwrite that row with the new data.

```
call DBGET (matrix)
  .
  perform some operations on the data contained in the “matrix” structure
  .
call DBPUT (matrix)
```

If it is desired to completely rewrite the "matrix" structure and use it to overwrite what is already in the database without having to read the row first, then a call to subroutine DBREPL is performed as such:

```
call DBREPL (matrix)
```

Only the first row that met the search criteria from the most recent call to DBFIND is overwritten. If additional rows are to be overwritten in this manner, it will be necessary to call DBFIND again specifying exactly which one row is to be overwritten with a subsequent call to DBREPL. For this reason, it is best to use the combination of DBGET and DBPUT to overwrite several rows of data within a relation, and use DBREPL when only one row of data exists for a given relation.

Note that only one relation may be opened at a time. Database subroutine calls that perform any read or write operation do so only on the relation most recently opened by a call to DBFIND. This is true even if the calls to DBFIND and other database subroutine calls occur in different user supplied subroutines. A call to DBADD also changes the currently opened relation, so DBFIND must be called again before attempting to read from the database.

Before the user's program terminates, a call must be made to DBCLOSE to properly close the database file and rewrite internal database pointers and variables.

```
call DBCLOSE
```

If this call is not made, or if the user's program terminates with an error before calling DBCLOSE, the database may become corrupted and some or all of the data will not be accessible. In general, only data that was written since the last call to DBCLOSE will be lost, but this is not guaranteed.

There are many other user callable utilities available in these database routines which are not mentioned in this section. Reference should be made to Appendix I for a complete description of each user callable subroutine in the database system.

Appendix I – Subroutine Definitions

This appendix describes each user callable subroutine in the DB90 system. It is up to the user to make sure that the number and type of arguments passed between these routines and the calling program are correct. The only exception to this requirement is that the data array representing the row of data to be read from or written to the database may be any argument type. These routines are presented here in alphabetical order, and not in the order in which they are typically called.

Subroutine DBADD

```
call DBADD (REL, K1, T1, K2, T2, K3, T3, K4, T4, K5, T5, DATA)
character(len=*), intent(in)     :: REL
character(len=*), intent(in)     :: K1, K2, K3, K4, K5
integer, intent(in)              :: T1, T2, T3, T4, T5
integer, intent(in)              :: DATA
```

This routine is called to add a new row of data to the database for this relation. The key names must correspond to those used to define this relation, but they can be in any order. It is advisable to assign a key value to every key name defined for this relation. If you do not, searching for a key value when retrieving rows of data may return unpredictable results. Arguments for all 5 keys must be passed to this routine, with the unused keys typically being passed as a blank one character string for the key name and a zero for the key value.

REL	Relation name, 1 to 12 characters in length.
K1…K5	Key names, 1 to 8 characters in length. These may be blank if not needed.
T1…T5	Key values. These are typically set to zero if not used with a corresponding key name.
DATA	Array containing the entire row of data for the relation.

Example:
```
call DBADD ('relname',           'A',  12,     &
                           'SPECIAL', 100,     &
                                 ' ',   0,     &
                                 ' ',   0,     &
                                 ' ',   0,     &
                  data)
```

Function DBADR

```
DBADR (DATA)
integer(LONG), intent(out)      :: DBADR
integer(LONG), intent(in)       :: DATA
```

This function is called to return the address of a Fortran variable as a long integer (8 bytes). It is used to assign array addresses to long integer variables to facilitate using allocatable arrays with the database. It will be necessary to declare this function as an external function to correctly define its type.

DATA	Fortran array whose address is desired.

Example:
```
integer(LONG), external :: DBADR

integer(LONG) :: address
address = DBADR(data)
```

Subroutine DBCHECK

```
call DBCHECK (REL, ROWS, ROWLEN)
character(len=*), intent(in)    :: REL
integer, intent(out)            :: ROWS
integer, intent(out)            :: ROWLEN
```

This routine is called to check on the status of a relation. If the relation has been defined, the number of rows already in the database, if any, and the length of each row, in bytes, is returned. If the relation has not been defined yet, the row length is set to zero.

REL	Relation name.
ROWS	The number of existing rows in this relation.
ROWLEN	The row length, in bytes.

Example:
call DBCHECK (‘relname’, rows, rowlen)

Subroutine DBCLOSE

call DBCLOSE

This routine is called to close the database. It must be called before the user’s program terminates or the database may become corrupted.

Example:
call DBCLOSE

Subroutine DBCREATE

```
call DBCREATE (NAME)
character(len=*), intent(in)      :: NAME
```

This routine is called to create a new database file and initialize the database pointers and structures. If a file with the same name already exists, it will be destroyed and replaced with the new empty database.

NAME	Filename for the new database.

Example:
call DBCREATE (‘filename’)

Subroutine DBDEL

call DBDEL

This routine is called to delete selected rows of a relation from the database. The relation and the rows to delete are selected by calling DBFIND.

Example:
call DBDEL

Subroutine DBFIND

```
call DBFIND (REL, K1, C1, T1, B12, K2, C2, T2, B23, K3, C3, T3, B34, K4, C4, T4, B45, K5, C5, T5)
character(len=*), intent(in)      :: REL
character(len=*), intent(in)      :: K1, K2, K3, K4, K5
character(len=2), intent(in)      :: C1, C2, C3, C4, C5
integer, intent(in)               :: T1, T2, T3, T4, T5
character(len=1, intent(in)       :: B12, B23, B34, B45
```

This routine is called to select an existing relation in the database and to choose a subset of the rows of that relation for subsequent reading and writing operations. The Boolean operators provide a means of selecting ranges of values for each key and for linking keys together. Since only one relation may be active at a time, calling this routine also closes the previously active relation. Arguments for all 5 keys must be passed to this routine, with the unused key names and Boolean operators typically being passed as blank one character strings and zeros being passed for the key values.

REL	Relation name, 1 to 12 characters in length.
K1…K5	Key names, 1 to 8 characters in length. These may be blank if not needed. The special key name ‘ROWS’ may also be used.
C1…C5	Key boolean operator, which may be one of: eq, ne, gt, lt, ge, le, mx, mn meaning equal, not equal, greater than, less than, greater than or equal to, less than or equal to, maximum, and minimum, respectively.
T1…T5	Key values. These are typically set to zero if not used with a corresponding key name.
B12	Boolean operator relating key 1 to key 2, which may be either ‘a’ for AND or ‘o’ for OR.
B23	Boolean operator relating key 2 to key 3, which may be either ‘a’ for AND or ‘o’ for OR.
B34	Boolean operator relating key 3 to key 4, which may be either ‘a’ for AND or ‘o’ for OR.
B45	Boolean operator relating key 4 to key 5, which may be either ‘a’ for AND or ‘o’ for OR.

Example:

```
call DBFIND (‘relname’, ‘SPECIAL’, ‘eq’,     4, ‘a’,       &
                           ‘ITEM’, ‘ge’, 127, ‘ ‘,       &
                                ‘ ‘, ‘ ‘,      0, ‘ ‘,      &
                                ‘ ‘, ‘ ‘,      0, ‘ ‘,      &
                                ‘ ‘, ‘ ‘,      0)
```

In this example, DBFIND will set internal parameters to point to a subset of the rows of relation ‘relname’ that have the ‘SPECIAL’ key equal to 4 AND where the ITEM key is greater than or equal to 127. A special key name, ‘ROWS’, may also be used to select a row or rows regardless of key values.

Subroutine DBGET

```
call DBGET (DATA)
integer, intent(out)              :: DATA
```

This routine is called to retrieve the next available row of the current relation. The relation and the subset of rows to read from must first be selected using DBFIND.

DATA	Array to receive the row of data.

Example:

```
call DBGET (data)
```

Subroutine DBGETK

```
call DBGETK (KEY, VAL)
character(len=*), intent(in)      :: KEY
integer, intent(out)              :: VAL
```

This routine is called to retrieve a key value matching the supplied key name from the most recently read row of the current relation. This routine lets you retrieve key values regardless of the criteria used to select the row of data. The row of data must have been previously read using DBGET.

KEY	Key name, 1 to 8 characters in length.
VAL	Key value.

Example:
call DBGETK ('SPECIAL', value)

Subroutine DBGETV

call DBGETV (VAR, DATA, INDEX)

character(len=*), intent(in)	:: VAR
integer, intent(out)	:: DATA
integer, optional, intent(in)	:: INDEX

This routine is called to retrieve a variable value matching the supplied variable name from the next available row of the current relation. If the variable is an array and the optional index argument is present, only that one array value is retrieved.

VAR	Variable name, 1 to 8 characters in length.
DATA	Variable or array to store the variable data into.
INDEX	Index into the array, if the variable is an array. If index is present, it must be a valid array index.

Example:
call DBGETV ('XYZ', data, index)
or
call DBGETV ('XYZ', data)

The first example retrieves only the one array element of the 'XYZ' array at the 'index', returning it into the variable data(1). The second example retrieves the entire 'XYZ' array into the array data.

Subroutine DBGKEYS

call DBGKEYS (KEY, VAL)

character(len=*), intent(in)	:: KEY
integer, intent(out)	:: VAL

This routine is called to retrieve all of the key values matching the supplied key name from the current relation. This routine lets you retrieve key values regardless of the criteria used to select the row of data. The relation and the subset of rows to read from must first be selected using DBFIND.

KEY	Key name, 1 to 8 characters in length.
VAL	Array of key values.

Example:
call DBGKEYS ('SPECIAL', values)

Subroutine DBINFO

call DBINFO (NROW, IROW, JROW)

```
integer, intent(out)          :: NROW
integer, intent(out)          :: IROW
integer, intent(out)          :: JROW
```

This routine is called to return the number of rows of the current relation that exist in the database matching the search criteria specified in the most recent call to DBFIND. One should use this routine to determine how many rows are available to read, and to check that there are in fact rows to read. This will avoid generating errors from attempting to read data that does not exist.

NROW	Number of data rows matching the search criteria.
IROW	Row counter.
JROW	Actual row number corresponding to IROW.

Example:
call DBINFO (nrows, irow, jrow)

Subroutine DBOPEN

```
call DBOPEN (NAME)
character(len=*), intent(in)  :: NAME
```

This routine is called to open an existing database. If the database does not already exist, an error is generated and the user's program will terminate. Databases must first be created using DBCREATE.

NAME	Filename of an existing database.

Example:
call DBOPEN ('filename')

Subroutine DBPUT

```
call DBPUT (DATA)
integer, intent(in)           :: DATA
```

This routine is called to replace the most recently read row of data from the current relation. The row must have been read first using DBGET.

DATA	Array containing the row of data.

Example:
call DBPUT (data)

Subroutine DBPUTK

```
call DBPUTK (K1, T1, K2, T2, K3, T3, K4, T4, K5, T5)
character(len=*), intent(in)  :: K1, K2, K3, K4, K5
integer, intent(in)           :: T1, T2, T3, T4, T5
```

This routine is called to replace the key values for the most recently read row from the current relation. The row must have been read first using DBGET. The main purpose of this routine is to change the values of keys for a row of data that has been changed and replaced using DBPUT. Arguments for all 5 keys must be passed to this routine, with the unused keys typically being passed as a blank one character string for the key name and a zero for the key value.

K1…K5	Key names, 1 to 8 characters in length. These may be blank if not needed.
T1…T5	Key values. These are typically set to zero if not used with a corresponding key name.

Example:

```
call DBPUTK (           'A',   12,      &
               'SPECIAL', 100,      &
                          ' ',    0,      &
                          ' ',    0,      &
                          ' ',    0)
```

Subroutine DBPUTV

```
call DBPUTV (VAR, DATA, INDEX)
character(len=*), intent(in)       :: VAR
integer, intent(in)                :: DATA
integer, optional, intent(in)      :: INDEX
```

This routine is called to store a variable value matching the supplied variable name to the most recently accessed row of the current relation. If the variable is an array and the optional index argument is present, only that one array value is stored. The row must have been previously retrieved by either DBGET or DBGETV.

VAR	Variable name, 1 to 8 characters in length.
DATA	Variable or array to read the variable data from.
INDEX	Index into the array, if the variable is an array. If index is present, it must be a valid array index.

Example:

```
call DBPUTV ('XYZ', data, index)
or
call DBPUTV ('XYZ', data)
```

The first example writes only the one array element of the 'XYZ' array at the 'index' to the database, getting it from the variable data(1). The second example writes the entire 'XYZ' array to the database.

Subroutine DBREPK

```
call DBREPK (K1, T1, K2, T2, K3, T3, K4, T4, K5, T5)
character(len=*), intent(in)       :: K1, K2, K3, K4, K5
integer, intent(in)                :: T1, T2, T3, T4, T5
```

This routine is called to replace the key values for the first row found from the most recent call to DBFIND. Use of this routine is similar to DBPUTK except the row of data does not need to be read first.

Arguments for all 5 keys must be passed to this routine, with the unused keys typically being passed as a blank one character string for the key name and a zero for the key value.

K1…K5	Key names, 1 to 8 characters in length. These may be blank if not needed.
T1…T5	Key values. These are typically set to zero if not used with a corresponding key name.

Example:

```
call DBREPK (           'A',   12,     &
                  'SPECIAL', 100,      &
                        ' ',    0,     &
                        ' ',    0,     &
                        ' ',    0)
```

Subroutine DBREPL

```
call DBREPL (DATA)
integer, intent(in)                :: DATA
```

This routine is called to replace the first row of data found from the most recent call to DBFIND. Use of this routine is similar to DBPUT except the row of data does not need to be read first.

DATA	Array containing the row of data.

Example:

```
call DBREPL (data)
```

Subroutine DBTOC

```
call DBTOC (NUNIT)
integer, intent(in)                :: NUNIT
```

This routine is called to write the table of contents of the currently open database to the specified Fortran file unit.

NUNIT	Fortran unit number (typically 6 for the standard Fortran output file) to write the table of contents to.

Example:

```
call DBTOC (6)
```

Subroutine DBVINFO

```
call DBVINFO (VAR, VLENG, VTYPE)
character(len=*), intent(in)       :: VAR
integer, intent(out)               :: VLENG
integer, intent(out)               :: VTYPE
```

This routine is called to retrieve the length and type of a variable for the currently opened relation.

VAR	Variable name, 1 to 8 characters in length.
VLENG	Variable length in words, except when the variable is a character then the length is in bytes. Word length is machine dependent but is the length of a standard integer or single precision real variable, usually 4 bytes.
VTYPE	Variable type.

Example:
call DBVINFO ('XYZ', vlength, vtype)

Subroutine DCADDR

```
call DCADDR (RNAME, KNAME, VNAME, VTYPE, VLENG, NK, NV)
character(len=*), intent(in)    :: RNAME
character(len=*), intent(in)    :: KNAME
character(len=*), intent(in)    :: VNAME
integer, intent(in)             :: VTYPE
integer, intent(in)             :: VLENG
integer, intent(in)             :: NK
integer, intent(in)             :: NV
```

This routine is called to define a new relation in the currently open database. Routine DBCHECK may be called first to determine if this relation has already been defined, if necessary. New relations may be added to the database at any time.

RNAME	Relation name, 1 to 12 characters in length.
KNAME	Array of key names, 1 to 8 characters in length. There can be at most 5 keys.
VNAME	Array of variable name, 1 to 8 characters in length. There can be at most 100 variable names in each relation.
VTYPE	Array of variable type identifies corresponding to the array of variable names. The identifiers are: 0 – integer 1 – single precision real 2 – double precision real 4 – character 10 – integer pointer 11 – single precision real pointer 12 – double precision real pointer
VLENG	Array of variable lengths corresponding to the array of variable names. The lengths refer to the number of words for integer and real variables, and to the number of characters for character variables.
NK	The number of key names, 0 to 5.
NV	The number of variables. There must be at least one variable in each relation.

Example:
```
kname(1) = 'SPECIAL'
kname(2) = 'ITEM'
nk = 2
vname(1) = 'MATRIX1'
vname(2) = ‘VECTOR1’
vname(3) = ‘NUM’
vname(4) = ITEM’
vtype(1) = 2
vtype(2) = 2
vtype(3) = 0
vtype(4) = 0
vleng(1) = 200 * 400
vleng(2) = 400
vleng(3) = 1
vleng(4) = 1
nv = 4
call DCADDR (‘relname’, kname, vname, vleng, nk, nv)
```

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)*	01- 02 - 2005
2. REPORT TYPE	Contractor Report
3. DATES COVERED *(From - To)*	
4. TITLE AND SUBTITLE	DB90—A Fortran Callable Relational Database Routine for Scientific and Engineering Computer Programs
5a. CONTRACT NUMBER	NAS1-00135
5b. GRANT NUMBER	
5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S)	Wrenn, Gregory A.
5d. PROJECT NUMBER	
5e. TASK NUMBER	
5f. WORK UNIT NUMBER	23-762-45-10
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	NASA Langley Research Center, Hampton, VA 23681-2199; Swales Aerospace, Hampton, Virginia 23681
8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	National Aeronautics and Space Administration, Washington, DC 20546-0001
10. SPONSOR/MONITOR'S ACRONYM(S)	NASA
11. SPONSOR/MONITOR'S REPORT NUMBER(S)	NASA/CR-2005-213524

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 61
Availability: NASA CASI (301) 621-0390

13. SUPPLEMENTARY NOTES
Langley Technical Monitor: William M. Kimmel
An electronic version can be found at http://ntrs.nasa.gov

14. ABSTRACT

This report describes a database routine called DB90 which is intended for use with scientific and engineering computer programs. The software is written in the Fortran 90/95 programming language standard with file input and output routines written in the C programming language. These routines should be completely portable to any computing platform and operating system that has Fortran 90/95 and C compilers. DB90 allows a program to supply relation names and up to 5 integer key values to uniquely identify each record of each relation. This permits the user to select records or retrieve data in any desired order.

15. SUBJECT TERMS
Fortran; Computer program; Database; Relational

16. SECURITY CLASSIFICATION OF: a. REPORT	b. ABSTRACT	c. THIS PAGE	17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON	19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	UU	20	STI Help Desk (email: help@sti.nasa.gov)	(301) 621-0390

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18