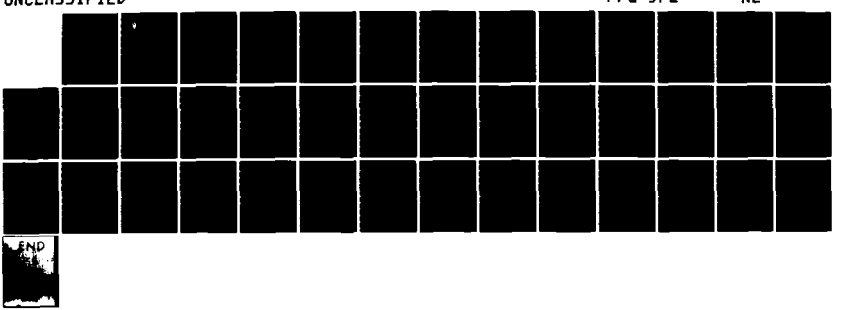
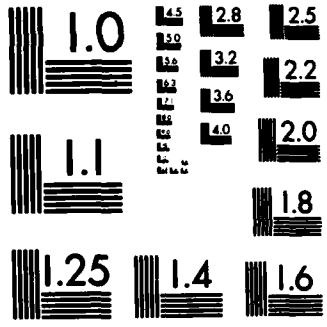
AD-A144 446 ADAM: AN ABSTRACT DATABASE MACHINE(U) ROYAL SIGNALS AND 1/1
RADAR ESTABLISHMENT MALVERN (ENGLAND)
N E PEELING ET AL. MAY 84 RSRE-84007 DRIC-BR-92291
UNCLASSIFIED F/G 9/2 NL





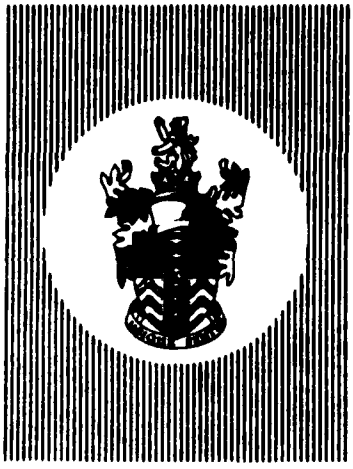
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

UNLIMITED

(2)

BR92291

Report No. 84007



Report No. 84007

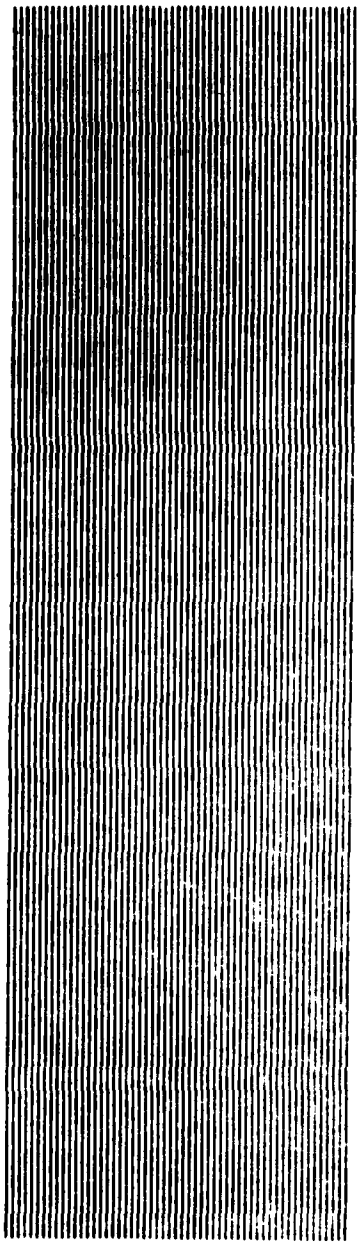
ROYAL SIGNALS AND RADAR ESTABLISHMENT,
MALVERN

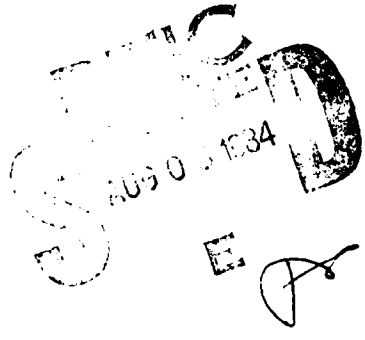
AD-A144 446

ADAM: AN ABSTRACT DATABASE MACHINE

Author: N E Peeling, J D Morison
E V Whiting

DTIC FILE COPY





PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE
RSRE
Malvern, Worcestershire.

May 1984

UNLIMITED

84 08 07 100

UNLIMITED

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Report 84007

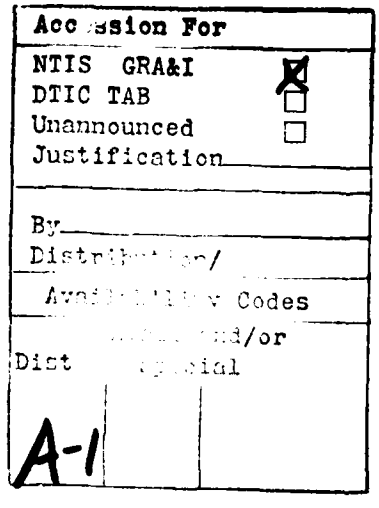
TITLE: ADAM: AN ABSTRACT DATABASE MACHINE

AUTHORS: N E Peeling, J D Morison, E V Whiting

DATE: May 1984

SUMMARY

ADAM extends a programming language to allow data on disc to be handled as easily as in-store data. It is a set of read/write and disc management procedures. These procedures work within standard random-access files in the filestore of the host operating system.



Copyright
C
Controller HMSO London

UNLIMITED

CONTENTS

1 INTRODUCTION

- ADAM extends a programming language to allow data on disc to be handled as easily as in-store data. It is a set of read/write and disc management procedures. These procedures work within standard random-access files in the filestore of the host operating system.

- Data of variable length is automatically allocated space in the file. Write procedures are of the form "here is some data, write it away and give me a pointer to it". Read procedures are of the form "here is a pointer, give me the data that it points to".

- ADAM is specifically designed to provide a high-integrity environment for data storage. This integrity does not rely on any features of the host operating system other than its ability not to corrupt its own filestore. ADAM has no operations that have to be used in a particular way to avoid compromising the integrity of the data in the files. ADAM is a non-overwriting system in which all data is ultimately accessible from a single pointer (called the root). Only the root can be assigned to. It is impossible to change the data that a pointer points to (this does not preclude the modification of the contents of the files, as will be explained later). This means that the data accessed from the root is completely unchanged until the root is changed. As a result the premature termination of a program (due to a program bug, or a machine crash, or a user induced break-in) cannot leave the file in a partially updated form.

- The data in the database is held as a tree structure. The root points to a block that can contain pointers to other blocks that can themselves contain pointers (and so on). To modify data the pointer chain to the place that needs changing must be recreated. To incorporate the change the root must be updated to include this new pointer chain. The whole tree structure may be contained in a single file within the host filestore. This file will be called the "master" file. Alternatively data can be directed to other files, called "slave" files. The division between the master file and the various slave files is completely under the control of the user of the ADAM package. The only restrictions are that the root must be contained in the master file and that all pointers within a slave file must point to data within that slave. The use of slave files allows the user to optimise the performance of the ADAM package on very large databases.

- Blocks of data can contain words, or characters, or pointers, or a mixture of words and pointers. These form the basic building blocks which can be built up into higher-level structures.

- A user-callable garbage collector is provided. This traces all data accessible from the root and frees any remaining space. This means that explicit deletion is unnecessary because data will effectively be deleted by the garbage collector whenever the pointer to it is no longer accessible from the root. The absence of an explicit delete operation means that there is no danger of deleting some data that is still being used (a very real danger when the same pointer is used in more than one place). The garbage collector is highly optimised so that it is usually feasible to call it interactively.

- A number of ADAM databases can be manipulated simultaneously.

- User documentation is available NOW.

- ADAM is already in use. Current applications are:

  - Implementation of a Design Management System for use by hardware designers.

  - A database to hold the software history of a mainframe computer system. It incorporates the inter-relationships between the different parts of the system.

- ADAM is highly portable. The current Algol68 version can be rehosted in less than a week.

- ADAM was designed for easy translation to other languages. Versions in Ada, Pascal and Modula2 could be done very easily. A slightly restricted version should be feasible in FORTRAN77 (under protest).

- Software written using ADAM can be made portable across all ADAM implementations.

The limitations of ADAM are:

- While the file is being written to, noone else may access it.

- Each individual random-access file will be limited to a maximum size which will be implementation dependent.

- There is an instore overhead that is a fixed percentage of the number of words in each random-access file being manipulated simultaneously. This percentage is implementation-dependent but most likely lies on the range (0.25 to 1)%. A slave file that is not in use does not incur this overhead.

- Because ADAM uses standard random access files it is not optimised for disc-access times (i.e. it makes no effort to make sure that data is contiguous on disc to reduce disc-head movements).

2 THE ADAM PACKAGE

The ADAM package consists of a number of procedures, plus some MODEs and constants.

All data in the database is held in arbitrary length blocks which are accessed by pointers. The MODE of a pointer is DISCPTR which is defined by ADAM. The user need know nothing further about this mode.

The internal workings of ADAM divide the random access files up into a number of equal sized portions, called "buckets". The user only need be aware of this fact when he is extending the size of his database (which can only be done in an integral number of buckets) or when he is particularly interested in using ADAM as efficiently as possible. Efficiency considerations are dealt with separately in section 3.

2.1 ADAM MODEs

```
DISCPTR  { This is the MODE of a pointer to a data block.  An improper
           DISCPTR is one that could not have been produced by ADAM
           for the database in use.  This includes unset DISCPTRs,
           DISCPTRs to blocks in other databases, DISCPTRs between
           slave files, and DISCPTRs from slave files to the master
           file of the database. }

MIXED = UNION(INT, DISCPTR)     { []MIXED is one of the types of block
                                  that can be kept in the database }
```

```
DISCFILE { This MODE is a structure that contains all the information
           that ADAM needs to control a database.  The fields are
           filled in when the connection is made to a database by a
           call of the procedure "getroot".  Whenever a procedure
           includes a DISCFILE as a parameter, that procedure will act
           on the database connected to that DISCFILE.  In this way
           any number of databases can be manipulated in one program.
           Some of the fields of DISCFILE contain information that the
           user must know about.  These are expanded below: }

         = STRUCT(DISCPTR root,      { the database root }
                  INT name,          { see the procedure "clear" }
                      filesize,      { the number of buckets in the
                                       binary (random access) file
                                       holding the master file of the
                                       database }
                      datecleared,   { an INT that represents the date
                                       that the procedure "clear" was
                                       last called }
                      secscleared,   { an INT that represents the time
                                       that the procedure "clear" was
                                       last called }
                      sizeoffreeblock, { the number of buckets free
                                         to be written to in the
                                         master file }
                      ......         { all other fields are hidden }
                  )
```

The size of the free block in a slave file and the filesize of a slave file can be found using the procedures "msizeoffreeblock" and "mfilesize" which are described in section 2.3.2.

2.2 ADAM constants

```
INT bucket size          { see the section on efficiency }

INT chars in bucket      {  "   "     "     "      "     }

INT useable              {  "   "     "     "      "     }

INT increase increment   { see the procedure "increase" }

INT max file size        {  "   "      "          "    }

INT min file size        { see the procedure "clear" }

DISCPTR nilptr           { a null pointer that can be written but
                           not read.  A procedure is provided to
                           check if a pointer is a "nilptr" }
```

2.3 ADAM procedures

2.3.1 Reading and writing procedures

The reading procedures do not deliver arrays of data as their results, for efficiency reasons. Instead the user must supply a reference to an array of the correct size to hold the data, and the read procedure will fill it in. For this reason ADAM supplies the following procedure:

```
PROC block elts = (REF DISCFILE df, DISCPTR dp)INT:
   { This procedure produces an INT that is the number of elements of
     data pointed to by "dp", in the database connected to "df". This
     can be used to declare an array variable of the correct size for
     use in a read procedure.

     possible failures:

     (i) "df" has not been initialised by a call of "getroot".

     (ii) "dp" is an improper DISCPTR.

     (iii) The data block is in a slave file which has not been made
           accessible by a call of "openslave".
   }


PROC write ints = (REF DISCFILE df, []INT dataout)DISCPTR:
   { This procedure writes "dataout" into the master file of the
     database connected to "df", and delivers a DISCPTR that points
     to it.

     possible failures:

     (i) You do not have write access to the master file of the
         database.

     (ii) "df" has not been initialised by a call of "getroot".

     (iii) There is not enough free space in the database; you must
           use "garbagecollect" or "increase".

     (iv) LWB "dataout" # 1
   }
```

```
PROC mwrite ints = (REF DISCFILE df,[]INT dataout,INT slaveno)DISCPTR:
   { This procedure writes "dataout" into the slave file number
     "slaveno" of the database connected to "df", and delivers a
     DISCPTR that points to it.

     Note that procedures which are concerned with the manipulation of
     multiple random access files (ie master/slaves) are prefixed by
     the letter "m" to distinguish them from procedures that act only
     on the master file.  If "slaveno" is zero then this is equivalent
     to calling the procedure on a master file.  The procedures which
     act by default on the master file (eg "writeints") are provided
     for upwards compatability reasons because earlier versions of
     ADAM only had master files.  They also provide simpler procedure
     calls for those users of ADAM who do not need to use slave files.

     possible failures:

     (i) You do not have write access to the slave file.

     (ii) "df" has not been initialised by a call of "getroot".

     (iii) The slave file number "slaveno" has not been made
           accessible by a call of "openslave".

     (iv) There is not enough free space in the database; you must
          use "garbagecollect" or "mincrease".

     (v) LWB "dataout" # 1
   }


PROC write chars = (REF DISCFILE df, []CHAR dataout)DISCPTR:
   { The same description as "writeints" }
```

```
PROC mwrite chars = (REF DISCFILE df,[]CHAR dataout,
                     INT slaveno)DISCPTR:
   { The same description as "mwriteints" }


PROC write discptrs = (REF DISCFILE df, []DISCPTR dataout)DISCPTR:
   { The same description as "writeints" but with one more possible
     failure:

     (v) "dataout" contains an improper DISCPTR.
   }


PROC mwrite discptrs = (REF DISCFILE df, []DISCPTR dataout,
                        INT slaveno)DISCPTR:
   { The same description as "mwriteints" but with one more possible
     failure:

     (v) "dataout" contains an improper DISCPTR.
   }


PROC write mixed = (REF DISCFILE df, []MIXED dataout)DISCPTR:
   { The same description as "writediscptrs" }


PROC mwrite mixed = (REF DISCFILE df, []MIXED dataout,
                     INT slaveno)DISCPTR:
   { The same description as "mwritediscptrs" }
```

```
PROC read ints = (REF DISCFILE df, REF[]INT datain, DISCPTR dp)VOID:
   { This procedure fills in "datain" with the data pointed to by
     "dp" in the database connected to "df".

     possible failures:

     (i) "df" has not been initialised by a call of "getroot".

     (ii) "dp" is a "nilptr".

     (iii) "datain" is the wrong size.

     (iv) LWB "datain" # 1

     (v) "dp" is a pointer to the wrong sort of data block.

     (vi) "dp" is an improper DISCPTR.

     (vii) "datain" is not contiguous.

     (viii) The data block is in a slave file which has not been
            made accessible by a call of "openslave".
   }


PROC read chars = (REF DISCFILE df, REF[]CHAR datain, DISCPTR dp)VOID:
   { The same description as "readints" }


PROC read discptrs = (REF DISCFILE df, REF[]DISCPTR datain,
                      DISCPTR dp)VOID:
   { The same description as "readints" }


PROC read mixed = (REF DISCFILE df, REF[]MIXED datain,
                   DISCPTR dp)VOID:
   { The same description as "readints" }
```

```
PROC can write = (REF DISCFILE df,
                  UNION([]INT, []CHAR, []DISCPTR, []MIXED)dataout)BOOL:
   { This procedure delivers TRUE if there is enough space to write
     "dataout" into the master file of the database connected to "df",
     and delivers FALSE if there is not.

     possible failures:

     (i) "df" has not been initialised by a call of "getroot".

     (ii) LWB "dataout" # 1
   }


PROC mcan write = (REF DISCFILE df, UNION([]INT, []CHAR,
                   []DISCPTR, []MIXED)dataout, INT slaveno)BOOL:
   { This procedure delivers TRUE if there is enough space to write
     "dataout" into the slave file number "slaveno" of the database
     connected to "df", and delivers FALSE if there is not.

     possible failures:

     (i) "df" has not been initialised by a call of "getroot".

     (ii) The slave file number "slaveno" has not been made accessible
          by a call of "openslave".

     (iii) LWB "dataout" # 1
   }
```

2.3.2 Database Management procedures

```
PROC clear = (INT channelno, name)VOID:
   { ADAM databases are just binary (random access) files that have
     been set-up in a particular way. This procedure sets-up the
     binary file that is connected to channel number "channelno" as
     the master file of the database. When procedure "getroot" is
     called, the root field of the DISCFILE parameter will be set to
     "nilptr" and the name field will be set to "name". The name
     field allows a user to distinguish his ADAM database from another
     person's.

     possible failures:

     (i) The file connected to channel number "channelno" is not a
         binary (random access) file.

     (ii) You do not have write access to the file connected to channel
          number "channelno".

     (iii) The file is not large enough to hold the information that
           ADAM needs to administer the database. The file must
           contain at least "minfilesize" buckets.

     (iv) Channel number "channelno" is already in use.
   }


PROC make slave = (REF DISCFILE df, INT channelno, slaveno)VOID:
   { This procedure sets-up the binary (random access) file that is
     connected to channel number "channelno" as the slave file number
     "slaveno" of the database connected to "df". Slave files must be
     made in numerical order starting with slave file number one.

     possible failures:

     (i)-(iv) The same as for PROC clear.
```

(v) "df" has not been initialised by a call of "getroot".

(vi) The slave file number ("slaveno"-1) has not had "makeslave" called on it.

}

```
PROC get root = (REF DISCFILE df, INT channelno)VOID:
```

{ This procedure connects the database whose master file is in the binary (random access) file connected to channel number "channelno" to the DISCFILE variable "df". It also gives access to the master file. When ADAM procedures have a DISCFILE as a parameter they will manipulate the database that was connected to that DISCFILE variable. This allows a number of ADAM databases to be manipulated simultaneously.

After the call of "getroot" the fields of the DISCFILE variable will all have been filled in. In particular, as the name of this procedure implies, the root field will then contain the root of the database connected to channel number "channelno".

There will be an instore overhead in the range (0.25 to 1)% of the size of the binary file connected to channel number "channelno".

possible failures:

(i) The file connected to channel number "channelno" is not a binary (random access) file.

(ii) The file does not contain the master file of an ADAM database ie it has not had "clear" called on it.

(iii) The file contains the master file of an ADAM database that was created by an old version of ADAM. It must be cleared again. This error should not occur once ADAM has passed out of the development phase.

(iv) ADAM has detected a potential corruption due to a machine crash during the last call of "finish". That "finish" did not take effect and to restart you must use "getlastroot" instead of "getroot".

(v) Channel number "channelno" is already in use.

(vi) The file connected to channel number "channelno" is a slave file.

}

```
PROC get last root = (REF DISCFILE df, INT channelno)VOID:
```

{ To be used instead of "getroot" in the case of failure (iv) in "getroot".

possible failures:

(i) The file connected to channel number "channelno" is not a binary (random access) file.

(ii) The file does not contain an ADAM database ie it has not had "clear" called on it.

(iii) The file contains an ADAM database that was created by an old version of ADAM. It must be cleared again. This error should not occur once ADAM has passed out of the development phase.

(iv) ADAM has not detected a potential corruption, you should have used "getroot".

(v) Channel number "channelno" is already in use.

(vi) The file connected to channel number "channelno" is a slave file.

}

```
PROC open slave = (REF DISCFILE df, INT channelno, slaveno)VOID:
   { This procedure gives access to the slave file number "slaveno" of
     the database connected to "df". The slave file is in the binary
     (random access) file connected to channel number "channelno".

     There will be an instore overhead in the range (0.25 to 1)% of
     the size of the binary file connected to channel number
     "channelno" for each slave opened.

     possible failures:

     (i) The file connected to channel number "channelno" is not a
         binary (random access) file.

     (ii) The file does not contain a slave file of an ADAM database
          ie it has not had "makeslave" called on it.

     (iii) The file contains a slave file that was created by an old
           version of ADAM. It must be made again. This error should
           not occur once ADAM has passed out of the development phase

     (iv) "df" has not been initialised by a call of "getroot".

     (v) Channel number "channelno" is already in use.

     (vi) The file contains a slave file but it is associated with a
          different database or has a different slave number.

     (vii) The file connected to channel number "channelno" is a
           master file.
    }
```

```
PROC check file = (INT channelno)INT:
   { This procedure checks for potential errors in "getroot".
     The INT result should be interpreted as follows:

          Result          Interpretation

            0        "getroot" should work satisfactorily.

            1        The file contains the master file of an ADAM
                     database that was created by an old version of
                     ADAM.  It must be cleared again.  This error
                     should not occur once ADAM has passed out of the
                     development phase.

            2        The file does not contain a master file ie it
                     has not had "clear" called on it.

            3        ADAM has detected a potential corruption due to a
                     machine crash during the last call of "finish".
                     That finish did not take effect and to restart
                     you must use "getlastroot" instead of "getroot".

            4        The file on channel number "channelno" is a slave
                     file.

     possible failures:

     (i) The file connected to channel number "channelno" is not a
         binary (random access) file.
   }
```

```
PROC check slave = (INT channelno)INT:
   { This procedure checks for potential errors in "openslave".
     The INT result should be interpreted as follows:

          Result          Interpretation

            0        "openslave" should work satisfactorily.

            1        The file contains a slave file of an ADAM
                     database that was created by an old version of
                     ADAM.  It must be made again.  This error should
                     not occur once ADAM has passed out of the
                     development phase.

            2        The file does not contain a slave file ie it has
                     not had "makeslave" called on it.

            3        The file on channel number "channelno" is a
                     master file.

     possible failures:

     (i) The file connected to channel number "channelno" is not a
         binary (random access) file.
   }


PROC finish = (REF DISCFILE df, DISCPTR root)VOID:
   { This procedure writes the "root" away into the database connected
     to the DISCFILE variable "df".  When "getroot" is next called on
     that database this new root will be fed into the root field of
     the DISCFILE variable that is the first parameter of "getroot".

     possible failures:

     (i) You do not have write access to the database ie to the master
         file and to any opened slave files.
```

(ii) "df" has not been initialised by a call of "getroot".

(iii) "root" is an improper DISCPTR.

(iv) There is not enough free space in the database to "finish". You must make more space by using "garbage collect", "mincrease" or "increase".

(v) The "root" does not point to a block in the master file of the database connected to "df".

}

```
PROC close slave = (REF DISCFILE df, INT slaveno)VOID:
```

{ This procedure makes the slave file number "slaveno" of the database connected to "df" inaccessible. It also frees the storage used in the overhead associated with "openslave". Any slave files which are not closed explicitly will be closed automatically by a call of "finish".

possible failures:

(i) You do not have write access to the slave file.

(ii) "df" has not been initialised by a call of "getroot".

(iii) The slave file number "slaveno" has not been made accessible by a call of "openslave".

(iv) There is not enough free space in the binary (random access) file to close it. You must make more space by using "garbagecollect" or "mincrease".

}

```
PROC can close = (REF DISCFILE df, INT slaveno)BOOL:
   { This procedure checks if there is enough space to "closeslave"
     number "slaveno" of the database connected to "df". It delivers
     TRUE if there is and FALSE if there is not. This procedure also
     can be used to check if there is enough space in slave number
     "slaveno" to allow "finish" to close that slave.

     possible failures:

     (i) "df" has not been initialised by a call of "getroot".

     (ii) The slave file number "slaveno" has not been made accessible
          by a call of "openslave".
   }


PROC can finish = (REF DISCFILE df)BOOL:
   { This procedure checks if there is enough space in the master file
     of the database connected to "df" to "finish". It delivers TRUE
     if there is and FALSE if there is not. There is no "mcanfinish"
     because "canclose" serves the same purpose.

     possible failures:

     (i) "df" has not been initialised by a call of "getroot".
   }


PROC mfilesize = (REF DISCFILE df, INT slaveno)INT:
   { This procedure delivers the number of buckets in the binary
     (random access) file holding slave file number "slaveno" of the
     database connected to "df".
```

possible failures:

(i) "df" has not been initialised by a call of "getroot".

(ii) The slave file number "slaveno" has not been made accessible by a call of "openslave".

}

```
PROC msizeoffreeblock = (REF DISCFILE df, INT slaveno)INT:
```

{ This procedure delivers the number of buckets free to be written to in the binary (random access) file holding slave file number "slaveno" of the database connected to "df".

possible failures:

(i) "df" has not been initialised by a call of "getroot".

(ii) The slave file number "slaveno" has not been made accessible by a call of "openslave".

}

```
PROC overhead = (REF DISCFILE df)INT:
```

{ This procedure delivers the number of buckets used by the ADAM system. These are not available for use in the master file of the database connected to "df" . The number of buckets available for the master file of the database is ("filesize" OF "df" - "overhead"("df")).

possible failures:

(i) "df" has not been initialised by a call of "getroot".

}

```
PROC moverhead = (REF DISCFILE df, INT slaveno)INT:
   { This procedure delivers the number of buckets used by the ADAM
     system in the slave file number "slaveno" of the database
     connected to "df".  The number of buckets available for the
     slave file is
     ("mfilesize"("df", "slaveno") - "moverhead"("df", "slaveno")).

     possible failures:

     (i) "df" has not been initialised by a call of "getroot".

     (ii) The slave file number "slaveno" has not been made accessible
          by a call of "openslave".
   }


PROC what is = (REF DISCFILE df, DISCPTR dp)INT:
   { This procedure tests to see what sort of data is in the block
     pointed to by "dp".  The INT result should be interpreted as
     follows:

     Result              Interpretation

       0          an improper DISCPTR

       1          "nilptr"

       2          pointer to a block of INTs

       3             "     " "   "    " CHARs

       4             "     " "   "    " MIXEDs

       5             "     " "   "    " DISCPTRs
```

```
    At the moment these are the only data blocks recognised.  ADAM
    is designed to allow for the easy addition of new types of data
    block.  This is of course a job for someone with a knowledge
    of the internal workings of ADAM.

    possible failures:

    (i) "df" has not been initialised by a call of "getroot".

    (ii) The block is in a slave file which has not been made
         accessible by a call of "openslave".
  }


PROC whereis = (REF DISCFILE df, DISCPTR dp)INT:
  { This procedure tells the user whether the block pointed to by
    "dp" is in the master file or a slave file.  If the result is 0
    the block is in the master file of the database connected to
    "df", otherwise the result is the number of the slave file
    containing the block.

    possible failures:

    (i) "df" has not been initialised by a call of "getroot".

    (ii) The block is in a slave file which has not been made
         accessible by a call of "openslave".

    (iii) "dp" is an improper DISCPTR.
  }
```

```
PROC increase = (REF DISCFILE df, INT by)VOID:
   { This procedure increases the size of the master file of the
     database connected to "df" by ("by" * "increase increment")
     buckets.

     possible failures:

     (i) You do not have write access to the binary (random access)
         file containing the master file.

     (ii) "df" has not been initialised by a call of "getroot".

     (iii) "increase" would take the size of the binary (random
           access) file beyond the limits set by ADAM.  The constant
           "max file size" gives the maximum number of buckets that
           can be accommodated.
   }


PROC mincrease = (REF DISCFILE df, INT by, slaveno)VOID:
   { This procedure increases the size of the slave file number
     "slaveno" of the database connected to "df" by
     ("by" * "increase increment") buckets.

     possible failures:

     (i) You do not have write access to the binary (random access)
         file containing the slave file.

     (ii) "df" has not been initialised by a call of "getroot".

     (iii) "mincrease" would take the size of the binary (random
           access) file beyond the limits set by ADAM.  The constant
           "max file size" gives the maximum number of buckets that
           can be accommodated.
```

(iv) The slave file number "slaveno" has not been made accessible by a call of "openslave".

}

```
PROC can increase = (REF DISCFILE df, INT by)VOID:
```

{ This procedure delivers FALSE if "increase" would fail due to an attempt to extend the binary (random access) file containing the master file past the limits set by ADAM, otherwise it delivers TRUE.

possible failures:

(i) "df" has not been initialised by a call of "getroot".

}

```
PROC mcan increase = (REF DISCFILE df, INT by, slaveno)VOID:
```

{ This procedure delivers FALSE if "mincrease" would fail due to an attempt to extend the binary (random access) file containing the slave file number "slaveno" past the limits set by ADAM, otherwise it delivers TRUE.

possible failures:

(i) "df" has not been initialised by a call of "getroot".

(ii) The slave file number "slaveno" has not been made accessible by a call of "openslave".

}

```
PROC connect slave = (REF DISCFILE df, INT channelno, slaveno)VOID:
   { This procedure marks the slave file number "slaveno" of the
     database connected to "df" for garbage collection, but does not
     give access to the slave.  This avoids the overhead inherent in
     "openslave".

     possible failures:

     (i) "df" has not been initialised by a call of "getroot".

     (ii) Channel number "channelno" is already in use.

     (iii) The file connected to channel number "channelno" does not
           contain a slave file ie it has not had "makeslave" called
           on it.
   }


PROC garbage collect = (REF DISCFILE df)VOID:
   { This procedure traces all blocks accessible from the root and all
     blocks written into the database connected to "df" since the last
     call of "finish" on "df".  Any other blocks are freed for use in
     the database.  Only those slave files which have previously been
     connected by a call of "openslave" or "connectslave" will be
     garbage collected.

     possible failures:

     (i) "df" has not been initialised by a call of "getroot".
   }
```

3 EFFICIENCY CONSIDERATIONS

There are two sorts of inefficiency that can be avoided if the user has some knowledge of the way ADAM works:

3.1 Efficient Useage of Space in the Database

ADAM divides the database into equal sized portions, called buckets. Each pointer points to an integral number of buckets, where the minimum size is one bucket. This means that if the database contains a large number of pointers to very small data blocks then there will be a lot of wasted space. The user can always pack a lot of small blocks into two larger blocks, where one block contains all the data and the other is an integer block that gives the break points within the large data block.

For example instead of having pointers to the CHAR blocks

"I F Currie"
"P W Edwards"
"J M Foster"
"J D Morison"
"N E Peeling"
"J Wood"

You could have the two blocks

"I F CurriеP W EdwardsJ M FosterJ D MorisonN E PeelingJ Wood"

(11, 22, 32, 43, 54, 60)

Alternatively the integers could give the length of the next character string

(10, 11, 10, 11, 11, 6)

Or, if the strings are all of less than a known length, a fixed number of characters could, by using a simple code, describe the number of characters in the next string. These coded characters could then be embedded in the CHAR block. This method needs to use just one block. For example think of the (silly) code where "a" stands for 1, "b" stands for 2 etc., and where all strings are less than 26 characters long then

"jI F CurriekP W EdwardsjJ M FosterkJ D MorisonkN E PeelingfJ Wood"

would do.

The tricks that can be played to pack small blocks are endless!

3.2 Efficient Useage of In-store Space

To write a data block away using the procedures in Section 2.3.1 it is necessary to produce all the data in-store before writing it into the database. A suite of procedures is provided for writing a data block "bit by bit". These procedures work in the form "here is a pointer to some data in the database and some data in-store that I wish to add to it, produce this larger data block in the database and give me a pointer to it". Each addition of this sort starts in a new bucket so it is much more efficient to write away a bucket full of data each time. Constants are provided (Section 2.2) to tell the user how many data elements fit into a bucket:

"bucket size"	INTs
"chars in bucket"	CHARs
"bucket size"	DISCPTRs
"useable"	MIXEDs

Procedures are also provided to read a data block, bucket by bucket; including procedures that tell the user how many buckets there are in a data block, and how many elements there are in each bucket.

```
PROC bucks in block = (REF DISCFILE df, DISCPTR dp)INT:
   { This procedure delivers the number of buckets in the data block
     pointed to by "dp", in the database connected to "df".

     possible failures:

     (i) "df" has not been initialised by a call of "getroot".

     (ii) "dp" is a "nilptr".

     (iii) "dp" is an improper DISCPTR.

     (iv) The data block is in a slave file which has not been
          made accessible by a call of "openslave".
   }


PROC buck elts = (REF DISCFILE df, DISCPTR dp, INT buckno)INT:
   { This procedure delivers the number of elements in bucket
     number "buckno" of the data block pointed to by "dp", in the
     database connected to "df".

     possible failures:

     (i) "df" has not been initialised by a call of "getroot".

     (ii) "dp" is a "nilptr".

     (iii) "dp" is an improper DISCPTR.

     (iv) The data block pointed to by "dp" does not have a bucket
          number "buckno".

     (v) The data block is in a slave file which has not been
         made accessible by a call of "openslave".
   }
```

```
PROC append ints = (REF DISCFILE df,
                    []INT dataout, DISCPTR dp)DISCPTR:
   { This procedure delivers a DISCPTR to a block that contains
     the data pointed to by "dp" (in the database connected to "df")
     with "dataout" appended at the end.

     possible failures:

     (i) You do not have write access to the binary (random access)
         file containing the data block.

     (ii) "df" has not been initialised by a call of "getroot".

     (iii) There is not enough free space in the database; you must
           use "garbagecollect", "mincrease" or "increase".  The
           procedures "mcanwrite" or "canwrite" tell whether there is
           enough space.

     (iv) LWB "dataout" # 1

     (v) The data block is in a slave file which has not been
         made accessible by a call of "openslave".
   }


PROC append chars = (REF DISCFILE df,
                     []CHAR dataout, DISCPTR dp)DISCPTR:
   { The same description as "appendints" }
```

```
PROC append discptrs = (REF DISCFILE df,
                        []DISCPTR dataout, DISCPTR dp)DISCPTR:
   { The same description as "appendints" but with one more
     possible failure:

     (v) "dataout" contains an improper DISCPTR.
   }


PROC append mixed = (REF DISCFILE df,
                     []MIXED dataout, DISCPTR dp)DISCPTR:
   { The same description as "appenddiscptrs" }


PROC buck of ints = (REF DISCFILE df, REF[]INT bucketin,
                     DISCPTR dp, INT buckno)VOID:
   { This procedure fills in "bucketin" with the data in the bucket
     number "buckno" of the data block pointed to by "dp", in the
     database connected to "df".

     possible failures:

     (i) "df" has not been initialised by a call of "getroot".

     (ii) "dp" is a "nilptr".

     (iii) "bucketin" is the wrong size.

     (iv) LWB "bucketin" # 1

     (v) "dp" is a pointer to the wrong sort of data block.

     (vi) "dp" is an improper DISCPTR.
```

(vii) "bucketin" is not contiguous.

(viii) The data block pointed to by "dp" does not have a bucket number "buckno".

(ix) The data block is in a slave file which has not been made accessible by a call of "openslave".

}

```
PROC buck of chars = (REF DISCFILE df, REF[]CHAR bucketin,
                      DISCPTR dp, INT buckno)VOID:
   { The same description as "buckofints" }
```

```
PROC buck of discptrs = (REF DISCFILE df, REF[]DISCPTR bucketin,
                         DISCPTR dp, INT buckno)VOID:
   { The same description as "buckofints" }
```

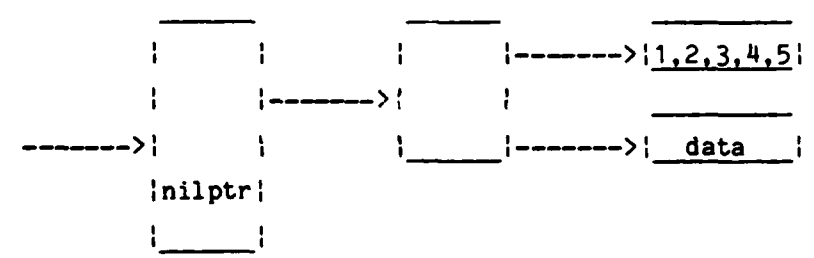
```
PROC buck of mixed = (REF DISCFILE df, REF[]MIXED bucketin,
                      DISCPTR dp, INT buckno)VOID:
   { The same description as "buckofints" }
```

4 EXAMPLES

The following sections contain some examples of how to create and manipulate an ADAM database:

4.1 Writing data to an ADAM database

A new ADAM database can be created as described below. In this example the root of the database points to a block of two DISCPTRs. The first DISCPTR points to another block of two DISCPTRs and the second is a nil pointer. In the second block of DISCPTRs the first DISCPTR points to a block of integers and the second to a block of characters. This particular database has no slave files associated with it.




```
INT name = ABS 8r12456321;  Name can have any value. It allows a user
                            to distinguish between his database and
                            other peoples by acting as a 24 bit
                            password.
INT channel = 1;            The binary file containing the database must
                            be connected to this channel.
DISCFILE df;
DISCPTR firstroot;
[1:2] DISCPTR mainptrs;
[1:2] DISCPTR dataptrs;
```

```
clear(channel, name);     The binary file connected to "channel" is
                          set-up as an empty ADAM database.
getroot(df, channel);     The database is connected to the DISCFILE
                          variable "df". The root field of "df" will be
                          a nilptr.

[1:5] INT integers := (1,2,3,4,5);
[1:4] CHAR characters := "data";
dataptrs[1] := writeints(df, integers);
dataptrs[2] := writechars(df, characters);
mainptrs[1] := writediscptrs(df, dataptrs);
mainptrs[2] := nilptr;
firstroot := writediscptrs(df, mainptrs);

finish(df, firstroot)     The next time "getroot" is called on the
                          database the root will be the DISCPTR
                          "firstroot".
```

4.2 Reading data from an ADAM database

To read and print the block of characters containing the string "data" (which was written away in section 4.1), the chain of pointers to the block of characters must be read in as shown below.

```
INT channel = 1;
DISCFILE df;

getroot(df, channel);     The root field of "df" will be the DISCPTR on
                          which "finish" was last called ie "firstroot".

[1:blockelts(df, root OF df)] DISCPTR mainptrs;
readdiscptrs(df, mainptrs, root OF df);

[1:blockelts(df, mainptrs[1])] DISCPTR dataptrs;
readdiscptrs(df, dataptrs, mainptrs[1]);
```

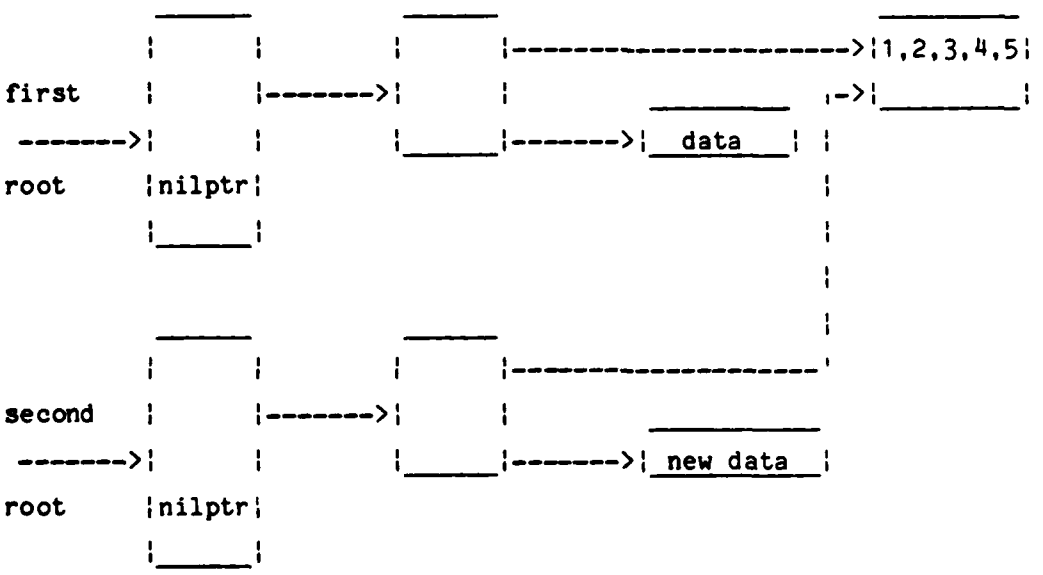
```
[1:blockelts(df, dataptrs[2])] CHAR characters;
readchars(df, characters, dataptrs[2]);

print((characters, newline))
```

"finish" is not needed since the database has not been altered.

4.3 Changing data in an ADAM database

The following is an example of how to change a block of data within the database created in section 4.1. In this example the block of characters "data" is replaced by "new data". Only the pointer chain to the new block of data needs to be updated so first the pointer chain is read in. The disc pointer to the block containing "data" is then replaced by a disc pointer to a new block containing "new data" and the pointer chain recreated as shown below. Any disc pointers to other blocks of data such as the integers are not altered.



```
INT channel = 1;
DISCFILE df;
DISCPTR secondroot;
[1:8] CHAR characters := "new data";

getroot(df, channel);

[1:2] DISCPTR mainptrs;
readdiscptrs(df, mainptrs, root OF df);

[1:2] DISCPTR dataptrs;
readdiscptrs(df, dataptrs, mainptrs[1]);
                        The pointer chain to the block of characters
                        has now been read in.

dataptrs[2] := writechars(df, characters);
                        The disc pointer to the new block of
                        characters is formed and replaces the old
                        pointer in "dataptrs[2]".
mainptrs[1] := writediscptrs(df, dataptrs);
secondroot := writediscptrs(df, mainptrs);
                        "secondroot" now contains the pointer chain to
                        the new data block.  Note that "mainptrs[2]"
                        and "dataptrs[1]" are unchanged.

finish(df, secondroot)  "finish" causes the new tree structure to
                        be stored in the database.
```

UNCLASSIFIED

5 ACKNOWLEDGEMENTS

The authors have pleasure in thanking I F Currie and J M Foster for all their help during the design of ADAM.

UNLIMITED

DOCUMENT CONTROL SHEET

Overall security classification of sheet ~~Unclassified~~

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference Report 84007	3. Agency Reference	4. Report Security Classification u/c
5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location Royal Signals and Radar Establishment		
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location		

7. Title

ADAM: AN ABSTRACT DATABASE MACHINE

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
PEELING, N E	MORISON, J D	WHITING, E V		

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

Unlimited

Descriptors (or keywords)

continue on separate piece of paper

Abstract ADAM extends a programming language to allow data on disc to be handled as easily as in-store data. It is a set of read/write and disc management procedures. These procedures work within standard random-access files in the filestore of the host operating system.

END

FILMED

DTIC