Segmented Data Files: an I/O Standard

William Symes

CRPC-TR91179
May, 1991

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Segmented Data Files: an I/O Standard

William W. Symes

Department of Mathematical Sciences

Rice University

Houston, Texas

Contents

Introduction

This document describes a file structure for disk storage of numerical data sets, a corresponding incore storage structure, and a means of translation between the two. The guiding principle of this standard is that *data files should contain within them all dimensional and other information necessary to interpret them properly.* The standard encompasses simple ASCII files, MATLAB `.mat` files, and many of the data formats used in geophysical data processing.

I wrote the procedures in C, with a Fortran interface so that the package is callable from either language. I provided an option for XDR encoding/decoding to generate portable binary archival files. I also provided a parallel set of routines for reading and writing `.tmp` files, which are byte-level images of the in-core structure. Since these `.tmp` files require no translation they are suitable for fast storage and retrieval of temporary datasets.

To use the package you need to:

1. understand the notion of *segmented data file*, and how such files are defined by *specification files*; build spec files for the disk and incore formats you wish to use, or borrow them from somewhere;
2. install the source and include files described below in your source directories;
3. insert appropriate calls in your code to open, read, write and close files using the procedures in the package.

The sections below take on each of these points in turn.

1 Segmented I/O

Data Files

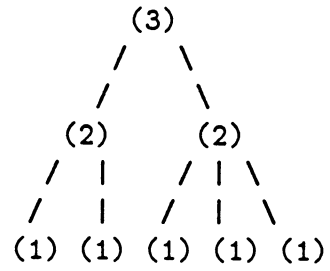
A segmented data file is a standard way of storing an array of numerical data broken up into subarrays. The defining characteristic of the standard is that the dimensional and other information needed to read and interpret the data is included in the file, along with the data, in the form of auxiliary subarrays called "headers". The headers are arranged heirarchically, so that information applying to a number of subarrays may be placed just once, before all of its subject data subarrays occur in the file.

A good example of this *segmented* file structure is the MATLAB `.mat` binary file, as described in the MATLAB manual (version 3.5) under "Load and Save". MATLAB places the information describing the matrix (the numbers of rows and columns, plus some other data necessary to load the matrix into MATLAB properly) at the beginning of the file. A matrix may be viewed as the collection of its columns, in order; every column has the same number of rows. To read a column, you need to know how many data words it contains, i.e. its size (= the number of rows). You can place that information either at the head of each column, or at the beginning of the file, since it applies to all of the columns. MATLAB chooses the second alternative. Of course, to read the matrix column-by-column you also need to know the number of columns. Once again, you could either place this information at the head of each column, or you could place it at the beginning of the file — either way, you can keep track of where you are in the file (matrix), and know when to stop, without relying on an EOF or other system-dependent device.

You could envision a file consisting of a number of matrices, perhaps of different sizes. Then you need to know the number of matrices in the file ("2D slices"), the number of columns in each matrix ("1D slices"), and the number of rows in each column ("size of 1D slice"). All of this information can be placed anywhere, so long as you read it *before* you need it. If the information is *local* to a given level or dimension of data, it can't be placed "higher" in the file. For example, if the columns have variable lengths (not allowed in MATLAB!), that information is local to 1D slices (columns) and

cannot sensibly be stored at the beginning of the file. The sizes of slices of various levels or dimensions comprise just some of the information which might be necessary to read, or properly interpret, a file. For example, to interpret *sampled data*, you need to know the sample rates or increments, so this information should be present in headers as well.

The organization of a file by level, or dimension, induces a tree structure, with the node at the top representing the entire file, nodes at all levels representing possible locations of headers, and nodes at the bottom representing the location of 1D (lowest level) headers and data slices. So a 3D (3 level) segmented file might be represented like this:



The numbers in parentheses designate levels (I use "dimension" as a synonym for "level" in this document, as it's appropriate for data sets generated by sampling functions of several variables on a regular, hypercubical grid).

In reading or writing a segmented file, you start at the top of the above picture, traveling down the left branch (say), collecting information from the headers as you go. You read data when you come the bottom (level 1). You can read slices at any level: for example, you could read the first 2D slice, composed of the first two 1D slices, then read the second 2D slice, composed of the last three 1D slices (in this example). As you read your way through the data, you move up a level in the tree and read the header at the next level, when you come to the last slice at the current level. For example, after reading the second 1D slice above, you must move up to level 2 and read the second header at that level, which should for instance tell that you have three 1D slices to read inside that 2D slice.

All of this is done without any worry on your part by the software. There are some restrictions, of course. Headers may have any length, including zero (i.e. no header at some level), but *all headers at the same level must have the*

same size and structure. That is, the headers (though not the data itself) form a *hypercubical* data set. The header information can be distributed almost arbitarily, except that by the time you get to the bottom of a branch you must have encountered all the information necessary to read the data. Also information local to a given level must be embedded in the header at that level or lower. In the example above, the number of 1D slices in a 2D slice can be stated in the header at level 1 or level 2, but not in the header at level 3. Header entries may have any numerical type (amongst those recognised by C — short, ordinary, or long integers, single and double precision floating point numbers), and numerical types of header entries can be mixed arbitrarily (but the mix is the same for every header at a given level). The data can have any numerical type, but only one numerical type is allowed in a given file type. For example all `.mat` files defined under the segmented i/o standard contain only double-precision data (which is MATLAB's default, though other numerical types are supported as non-default choices by MATLAB). At some point in the future it may become desirable to allow C structures as header or data items, but at present I haven't done so (with one exception for MATLAB, as explained below).

On disk, each file occupies (more or less) a linear block of storage. For flexibility, for use in a future database project, and to accomodate some existing formats (notably SEGY), each file begins with a *text block* of size $\geq$ 0 bytes. As explained below, each file type defined under the standard has a definite dimension *d*. Each k-dimensional slice, k = 1, ..., d, has a *header* the entries of which may have any numerical type (short, long, float, double). The 1D slices (*traces* after the geophysical usage) consist of header segments followed by data segments, thus:

1D header	data (n_1 samples)

The data may have any numerical type, but all data samples in the file must have the same numerical type.

A 2D slice (*record*) thus looks like:

2D hdr	trace 1	trace 2	...	trace n_2

and a 3D slice like

3D hdr	record 1	record 2	...	record n_3

The file looks like

text block	dD hdr	$d-1$ slice 1	...	$d-1$ slice n_d

Specification Files

In order to accomodate a number of existing formats (MATLAB `.mat` files, SEGY standard) I chose to encode the *file type name* in a suffix to the file name, rather than inside the file itself. Thus each segmented data file name has the form

`name.type`

where by convention the prefix `name` has no embedded blanks or periods. The suffix `type` is the file type name, and must be the name of a *specification ("spec") file* containing the type specification.

The method I've developed for defining file types under the segmented i/o standard is really a small high level language, with extremely limited syntax and vocabulary. Specification files completely specify a file type implemented under the standard, as well as the information needed to translate it into in-core representation. All specification files should be located in a "defaults" directory, the full pathname of which is given in "`#define`" statement in the file `iodefine.h` (more on this in the Installaation section, below). Each piece of information necessary for correct read/write and interpretation of a record file of type `type` must either be coded in the specification file `type` explicitly, or `type` must tell the i/o procedures where to look for it in the various headers. Three types of information must be present in a specification file:

- dimensional information necessary to read/write the file: the data dimension d, and the size of the text block in bytes (this may be fixed of length ≥ 0 bytes, or variable, as explained below);

- information necessary to type correctly the headers and data: the numerical type of each header word and the numerical type of the data samples;

- the *meanings* of header words, i.e. the assignment of these words to header entries in the in-core structure, and to the remaining items (particularly the sizes `datasize[k]` of the k-dimensional slices) necessary for i/o.

The last item (in-core meanings of out-of-core header entries) is coded by means of *specification strings*, which are intelligible both to the programmer and to the i/o procedures In fact the entire specification file stucture is designed to be human-readable.

Some items *must* be present in the specification file, others are optional. The syntax of the statements is simple and mostly plain English; assignments of numerical or character value are indicated with "=", whereas a colon separates an item from the statement of its header location (in effect a pointer). The order in which the various items are listed is immaterial (but may affect readability).

- Mandatory header entries:

  1. `encoding = ascii, binary or xdr`
  2. numerical type of data samples, like so:
     `data type = <type>` where `<type>` is one of the strings: `short, int, long, char, float, double`
  3. size of text block: either the character string `variable` or the character string `fixed`; in the latter case followed by
  4. `length of text block = <length in bytes>`
  5. data dimension;
  6. numerical types of each header entry, in sequence, like so:
     `type: dimension k entry j = <type>`

7. header locations of the sizes `datasize [k]` in each dimension k (note that in order to read the file, `datasize [k]` must be located in the header at level $\geq k$); locations are coded in English, like so: `size k: dimension j entry i`

- Optional header entries:

  These are all in the form of variable names followed by location pointers as in the last example above:
  `variablename: dimension k entry j`

Example

A simple 3D file structure along the lines of the tree example depicted above, with space for a variable-length comment section, would be specified by the following specification file:

```
data dimension = 3
encoding = ascii
size of text block = variable
data type = float
type: dimension 3 entry 1 = int
type: dimension 2 entry 1 = int
type: dimension 3 entry 2 = float
type: dimension 1 entry 1 = int
type: dimension 2 entry 2 = float
type: dimension 1 entry 3 = float
type: dimension 1 entry 4 = float
type: dimension 1 entry 5 = float
type: dimension 1 entry 6 = float
type: dimension 1 entry 7 = float
size 1: dimension 1 entry 1
size 2: dimension 2 entry 1
size 3: dimension 3 entry 1
number of shot records: dimension 3 entry 1
number of samples per trace: dimension 1 entry 1
sample interval: dimension 1 entry 2
```

```
trace offset: dimension 1 entry 3
number of traces per shot: dimension 2 entry 1
shot location: dimension 2 entry 2
```

This example illustrates several important features. All of the mandatory entries are present. The text block is specified as having variable length; when it is read or written, its end will be signified by the string #. Several of the optional header entries are in fact the same as some of the mandatory ones, but are assigned to additional specifications. These specifications connect the disk file header entries to in-core header entries by the i/o translation procedures, as described below. Thus it is presumed that the specifications listed are amongst those passed to the i/o procedures by the calling program, together with the corresponding indices in the in-core headers. Eight optional header entries are defined by specification strings. Several more header entries at dimension 1 (i.e. trace header entries) are declared but not associated with a specification. That is, these represent unused header slots. Note that the list of header declarations implicitly determines the length of each header in bytes, so it is important that every header entry actually present in the file be correctly declared, whether it is assigned a meaning in the current file type definition or not. Also note that several file types could use the same header declaration list, assigning specifications to various subsets of the available header entries.

For the reader's amusement, here is the spec file for the MATLAB `.mat` structure:

```
data dimension = 2
encoding = binary
size of text block = fixed
length of text block = 0
data type = double
type: dimension 2 entry 1 = mattype
type: dimension 2 entry 2 = int
type: dimension 2 entry 3 = int
type: dimension 2 entry 4 = int
type: dimension 2 entry 5 = matstring
size 1: dimension 2 entry 2
```

```
size 2: dimension 2 entry 3
MATLABtype: dimension 2 entry 1
mrows: dimension 2 entry 2
ncols: dimension 2 entry 3
```

Note that there are no level 1 header entries.

Because **mat** files don't quite fit the picture developed so far, I had to specify two special types of header entry. **mattype** is an int with always the same value for a given class of machine (in case of Suns and others with Motorola byte-order, double precision data, the value must always be 1000). It is ignored on reads and always written with the correct value on writes. This seemed reasonable because I always use segmented i/o in conjunction with MATLAB on Suns, but is obviously a kludge. Also present in the MATLAB 2D header are a string (the name MATLAB assigns to the data array when it's loaded) and its length in bytes (inlcuding the trailing NULL). I have also hard-coded this item as an informal "structure" **matstring**. Compare the description of **.mat** file structure in the MATLAB manual.

If such exceptions arise in the future it will be possible to accomodate them in the code in a less ad-hoc way, but it didn't seem worth it at the moment.

Incore Data Structure

A very important part of the segmented approach to i/o is that *incore data sets are also stored as segmented "files", with all relevant dimensional and interpretive information in header segments.* To simplify the manipulation of incore data sets, I have elected to store all incore data - both header entries and data samples - as a single numerical type. Currently I use single precision floating point numbers (i.e. real or real*4 in FORTRAN, float in C). The format for incore storage is given by another specification file, similar in form to those specifying disk files, named **incore** and stored in the same **defaults** directory as the other specification files. The formats common in exploration seismology motivated the definition of the in-core style: it is highly redundant, with all of the header information (data dimensions, sample rates, and the like) collected in trace (1D slice) headers, rather than

spread in an arbitrary way amongst the dimensions. This redundant design simplifies the coding of out-of-core procedures, since all necessary information is available in every record (in fact, in every trace!).

Thus each incore record looks exactly like an out-of-core record (2D slice - see above), with a zero-length 2D header.

The `incore` specification file is application-dependent: it is a list of descriptive strings together with their indices in the incore trace header. It defines the interface between arbitrary segmented disk files and the incore structure used in a particular application. On each read transaction, the incore header entries are collected from the various headers of the disk file, and conversely on each write transaction.

Here is an example of an `incore` file suitable for use with the simple disk file type for storage of a 3D data cube explained in the previous section:

```
number of shot records: 1
number of traces per shot: 2
number of samples per trace: 3
sample interval: 4
trace interval: 5
shot interval: 6
shot location: 7
trace offset: 8
```

NOTE: this is NOT the `incore` file contained in the sample directory `export_io/data`. The `incore` file included there is instead a longer one which I developed for representing seismic shot order data.

Programs using this standard should also be provided with a means to associate a parameter to each header index, uniformly throughout the program. At the moment, I write FORTRAN codes using this standard, so a convenient way to associate parameters to the header indices is through and "include" file containing a FORTRAN parameter statement. One that would work for the in-core structure just defined would look like this:

```
integer   nshots, ntraces, nsamples, tsample,
&            xtraceint, xshotint, xshot, x
```

```
&            lennheader
 parameter  (nshots        =  1,
&                ntraces       =  2,
&                nsamples      =  3,
&                tsample       =  4,
&                xtraceint     =  5,
&                xshotint      =  6,
&                xshot         =  7,
&                xtrace        =  8,
&                lenheader     =  8)
```

Note the inclusion of the variable `lenheader` giving the number of header entries in each (incore) trace. This information must be explicitly available in order to do memory offset calculations: the length of each in-core trace segment is `lenheader` + the number of data samples (i.e `datasize[1]`).

A VERY IMPORTANT REMARK: note that the files `tracehdr.h` and `incore` MUST be consistent: if you change one, you must change the other. I have gotten into trouble this way, so watch out.

If all code making use of the incore structure were written in C, a neater solution would be possible: the functions of the two files just described could be combined into a single C header file. Then the possibility of inconsistent definitions of incore header entries would be avoided in the nicest possible way. For example, in ANSI C the information in the two files just listed is contained in the following `incore.h` file:

```
const int nshots      1   /* number of shot records */
const int ntraces     2   /* number of traces per shot */
const int nsamples    3   /* number of samples per trace */
const int tsample     4   /* sample interval */
const int xtraceint   5   /* trace interval */
const int xshotint    6   /* shot interval */
const int xshot       7   /* shot location */
const int xtrace      8   /* trace offset */
const int lenheader   8   /* trace header length (words) */
```

This file can be read as a text file to connect the specification strings (i.e. the comments) to the numerical values of the indices, and used also as

an include file wherever the parameter names (now treated as *C* symbolic constants) are needed. This is doubtless the right long-term solution, but I don't see how to implement it cross-linguistically while keeping the indices as parameters (thus safeguarding against inadvertant assignments).

At the moment, Fortran/C incompatibility forces the use of a parallel trace header file for the C routines, declaring the small part of the incore trace headers needed for C i/o as C symbolic constants. I have taken to calling the Fortran trace header file `tracehdr.h`, and its C analogue `tracehdr_c.h`.

2 Installation

Somewhere in your source directories you should place the directory `export_io` (see me for full path name), which contains:

- `io.c`, which contains the basic i/o routines;
- `io_tmp.c`, which contains the `.tmp` i/o routines;
- `io.h` `tracehdr_c.h` and `iodefine.h`, necessary include files for the C part of the package;
- `tracehdr.h`, `ram.h`, and `system.h`, necessary include files for the Fortran interface;
- `io_f.c` and several `*.f` files, which constitute the Fortran interface;
- `copy.c`, a main program file for copying segmented files;
- `fcopy.f`, its FORTRAN analogue;
- a makefile.

You must decide where to keep your spec files, i.e. make a directory, and set the enviroment variable SEG_DEFAULTS to reflect your choice. That is, you must issue the command

`setenv SEG_DEFAULTS "full path name for spec file directory"`

This line is probably best added to your `.cshrc` file in your home directory.

The directory `export_io` contains a `data` subdirectory, which includes some example spec files and data sets. I keep my spec files in a separate directory.

Note that `io_f.c` and `io_tmp.c` come in two flavors. In one the procedure names are declared in caps (`_cap`) and in the other the procedure names are declared with appended underbars (`_udb`). This is to allow various systems to implement Fortran/C calls; for example the Suns want an underbar, while the Stardent demands caps. More modifications may be necessary; for example in Vax Fortran, C routine name must have an underbar *prepended* in calls from Fortran.

You must compile these routines, and stow the object files somewhere useful. I like to make a user library to keep them in. Here is a Makefile fragment which would work on Sun4 machines, and is included in `export_io`. It also makes the programs `copy` and `fcopy`.

```
.IGNORE:

F_SUBS = frecio.f
C_SUBS = io.c io_f_udb.c io_tmp_udb.c
FC     = f77 -u -cg89 -c
CC     = cc -cg89 -c
IOLIB  = iolib.a
BLAS   = /usr/local/lib/libblas.a

iolib: $(F_SUBS) $(C_SUBS)
        $(FC) $(F_SUBS)
        $(CC) $(C_SUBS)
        ar vr iolib.a *.o
        ranlib iolib.a
        rm *.o

copy: copy.c iolib.a
        cc -o copy copy.c $(IOLIB)
        mv copy data
```

```
fcopy: fcopy.f iolib.a
        f77 -u -cg89 -o fcopy fcopy.f $(IOLIB) $(BLAS)
        mv fcopy data
```

This makefile assumes that you've stowed the include files in the same directory as the source files, and that the "make" is running there.

3 Using Segmented I/O

The essential operations are opening, closing, reading, and writing files with the package. These can be accomplished from either Fortran or C. I will cover Fortran first.

The Fortran Interface

I have provided Fortran interface routines that read and write *records*, i.e. (nominally) 2D slices. This does not prevent you from reading and writing 1D data sets: just as a vector may be viewed as either a column vector or an $(n \times 1)$ matrix, a 1D data set may be viewed as a 2D data set with only one 1D slice. It does prevent you from reading and writing data with 1D slices of varying length.

To use any of the Fortran routines in this package you must first call `IOINIT`, which initializes variables and arrays used throughout the package. For example, the provision of unit numbers is handled automatically, and a table of current unit numbers is kept in a common block in `system.h` and initialized by `IOINIT`.

To open a file for reading or writing, you insert a call to `FILEOPEN`, the comment section of which is displayed here:

```
c=================================================
c SUBROUTINE FILEOPEN
c
```

```
c Purpose: opens data files for either native binary image
c of in-core data, in-core storage as a file image,
c archival segmented binary i/o, HDR header definition,
c or CFL (Rice Geophysics) unformatted Fortran i/o.
c Must be preceded by a call to SYSTSET.
c
c Arguments
c   name     -->  Filename.   MUST be in the form 'name.type',
c                 where 'type' is one of:
c                   'cfl' = Rice Geophysics (FORTRAN unformatted);
c                   'hdr' = current in-core header file (ascii);
c                   'ram' = in-core file;
c                   'tmp' = native binary image of incore record
c     {any other string} = one of the spec files in the directory
c                          reserved for such things; defined by
c                          a macro in SRC/GENERIC/H/IODEFINE.H
c                          (for archival segmented i/o in C).
c   iounit   <--  returns unit number (file pointer)
c   rw       -->  1 = read; 2 = write
c   ier      <--  error flag; = 0 for normal completion
c
c=====================================================
c
      subroutine fileopen(name,iounit,rw,ier)
c
```

Thus you pass the name of the file to be opened (`name`), and the read/write flag `rw`, = 1 for a read and = 2 for a write. You get in return a "unit number" to which the file has been attached. Note that this is only a real Fortran unit number if you have opened a CFL file (which occurs through the Fortran i/o system). Otherwise you have opened a file through the C i/o system, and `iounit` is actually an index into an array of file pointers or descriptors. Moral: *don't use* `iounit` *as a Fortran unit number; only use it in connection to other calls to the segmented i/o package.*

To read a record from the file opened for reading (`rw` = 1) on `iounit` into the array `buf`, you use FGETREC:

```
c=======================================================
c
c  SUBROUTINE FGETREC
c
c  WWS, 30.4.90
c
c  iounit     --->       unit number in segmented i/o system
c  ntotal     --->       total storage available in array {buf}
c  buf        <-->       array into which record is to be read
c  ier        <---       error flag; = 0 on successful completion
c
c  Purpose: extracts a record from a file, writes
c  it to the buffer {buf}.
c
c  Use: must be preceded by a call to FILEOPEN,
c  to open the file appropriately. Should be followed
c  (when all data is read) by call to FILECLS.
c
c=======================================================
c
        subroutine fgetrec(iounit, ntotal, buf, ier)
c
```

The input ntotal is included to safeguard against overwriting array boundaries. On return, ntotal = number of real*4's read.

To write a record of from the array buf to the file opened for writing (rw = 2) on iounit, use FPUTREC:

```
c=======================================================
c
c  SUBROUTINE FPUTREC
c
c  WWS, 30.4.90
c  revised 31.8.90
c  revised 20.2.91
c
c  Purpose: writes a record of ntotal words from the buffer
```

```
c  buf to the file at unit number iounit.
c
c=====================================================
c
        subroutine fputrec(iounit, ntotal, buf, ier)
c
```

On return, `ntotal` is the number of words written. The actual write is governed by the header structure in the array `buf` — i.e. `buf` cannot be merely a data array; it must actually be a 2D slice in the incore structure. Finally, you can win the Good Housekeeping seal of approval by politely closing your files when you're done. Amongst other things, this frees up the unit number `iounit` for re-use.

```
c
c=====================================================
c SUBROUTINE FILECLS
c
c Purpose - gee, I don't know, let me guess.
c
c=====================================================
c
      subroutine filecls(iounit,ier)
c
```

An Example: FCOPY

The modules described in the previous section are all used in the program `fcopy`, which is the FORTRAN analogue of the program `copy`, and is used in exactly the same way:

```
fcopy <source filename> <target filename>
```

Study of this source file should clarify the use of the FORTRAN interface.

```
c
c FCOPY: Fortran record-based copy using Segmented I/O
c
c--------------------------------------------------------------------
c
      program fcopy
c
c workspace for command line query
c
      integer numargs, iargc
c
c filenames, lengths, unit numbers, read/write flag
c
      character*80 name_in, name_out
      integer size_in, size_out, unit_in, unit_out, rw
c
c buffer for data, length
c
      integer max, ntotal
      parameter (max=1000000)
      real buf(max)
c
c max number of records permitted
c
      integer maxrec
      parameter (maxrec=300)
c
c error flag, record counter, query reply, verbosity flag
c
      integer ier, irec, iverb
      character yn
c
c include files
c
      include 'system.h'
      include 'tracehdr.h'
c
```

```
c-----------------------------------------------------------------------
c
      iverb = 1
      ier = 0

      call ioinit(ier)
      if (ier.ne.0) then
         write(ipout,*)' Error: FCOPY from IOINIT'
         stop
      end if

      numargs = iargc()

      if (numargs.ne.2) then
         write(ipout,*)' usage:'
         write(ipout,*)' fcopy <source filename> <target filename>'
         stop
      end if

      call getblank80(name_in)
      call getblank80(name_out)

      call getarg(1, name_in)
      call getarg(2, name_out)

      size_in  = index(name_in,' ') - 1
      size_out = index(name_out,' ') - 1

      write(ipout,*)' copy ',name_in(1:size_in),
     &              ' to ',name_out(1:size_out),' ? (y/n)'
      read(ipin,*)yn

      if (yn.eq.'y') then
c
c *** open name_in
c
      rw=1
```

```
      call fileopen(name_in,unit_in,rw,ier)
      if (ier.ne.0) then
         write(ipout,*)' Error: FCOPY from FILEOPEN'
         write(ipout,*)' file: ',name_in(1:size_in)
         write(ipout,*)' ier = ',ier
         stop
      end if
c
c *** open name_out
c
      rw=2
      call fileopen(name_out,unit_out,rw,ier)
      if (ier.ne.0) then
         write(ipout,*)' Error: FCOPY from FILEOPEN'
         write(ipout,*)' file: ',name_out(1:size_out)
         write(ipout,*)' ier = ',ier
         stop
      end if

      do 500 irec=1,maxrec

      if (iverb.eq.1) then
         write(ipout,*)' FCOPY: record = ',irec
         write(ipout,*)' FCOPY ---> FGETREC'
      end if

      ntotal = max
      call fgetrec(unit_in, ntotal, buf, ier)
      if (ier.ne.0) then
         write(ipout,*)' Error: FCOPY from FGETREC'
         write(ipout,*)' ier = ',ier
         stop
      end if

      if (iverb.eq.1) then
         write(ipout,*)' FCOPY ---> FPUTREC'
      end if
```

```
      call fputrec(unit_out, ntotal, buf, ier)
      if (ier.ne.0) then
         write(ipout,*)' Error: FCOPY from FPUTREC'
         write(ipout,*)' ier = ',ier
         stop
      end if
c
      if (irec.ge.maxrec) then
         write(ipout,*)' You cretin! You attempted to'
         write(ipout,*)' play BLAS with more than ',maxrec
         write(ipout,*)' records at once. That''s sheer,'
         write(ipout,*)' unmitigated greed, and you ought'
         write(ipout,*)' to be ashamed of yourself. I''m '
         write(ipout,*)' closing all of your files RIGHT NOW!!!'
         go to 300
c
c *** NOTE THE (TYPICAL) USE OF A HEADER ENTRY AS DIMENSIONAL
c     INFORMATION IN THE FOLLOWING STATEMENT, WHICH CHECKS TO
c     SEE IF WE'VE JUST COPIED THE LAST RECORD:
c
      else if (irec.ge.nint(buf(nshots))) then
         go to 300
      end if
c
500   continue
c
300   continue
c
      call filecls(unit_in,ier)
      call filecls(unit_out,ier)
      if (ier.ne.0) then
         write(ipout,*)' WATCH OUT! one of your files didn''t close'
      end if

      end if
```

```
      stop
      end
c
c**********************************************************************
c
      subroutine getblank80(blank80)
      character*80 blank80
      integer i
      do 1 i=1,80
         blank80(i:i)=' '
 1    continue
      return
      end
```