EDITOR USER MANUAL

Version 2.1

David A. Clarke

Institute for Computational Astrophysics

Saint Mary's University

Halifax NS, Canada B3H 3C3

`http://www.ica.smu.ca/zeus3d`

June, 2004; revised October, 2007

Contents

Disclaimer

EDITOR USER MANUAL

Version 2.1, David A. Clarke, ICA, October 2007

1 Introduction

1.1 VERSION 2.1

EDITOR is a highly portable text manipulator written in FORTRAN77 designed to manage and compile large computer codes, and placed in the public domain by the author (see *Disclaimer* on page *ii*). The "tar" file `dzeus35.tar.gz`, available for downloading at `www.ica.smu.ca/zeus3d`, contains the *EDITOR* package including this manual and installation intructions necessary to install and use `EDITOR` on the user's home platform.

EDITOR is designed for source code written in FORTRAN, although some of its functionality is independent of the contents of the ASCII text it manipulates. It was born, in part, out of the author's frustration in porting software from CTSS to UNICOS in 1988 when the NCSA switched the operating system on its Cray machines. For those familiar with the CTSS environment, *EDITOR* was initially designed to mimic the CTSS precompiler, *HISTORIAN*, much of whose functionality was not carried into *MPPL*, the first precompiler under UNICOS. Since then, *EDITOR* has blossomed into a rather sophisticated package in excess of 13,000 lines of FORTRAN capable of a variety of text manipulations. Each of these functionalities shall be referred to as a *job*, of which there are seven in this release. These include:

1. precompiling source code, including inserting modules (*e.g.*, common block declarations) into source code, selecting source code to be compiled, replacing `namelist` statements and their associated reads/writes with calls to subroutines in a portable library, " micro-tasking" nested do-loops, and splitting up the source code into modules (`PRECOM`, §2);

2. generating a multi-columned source code listing complete with a table of contents (`NUMBER`, §3);

3. merging a "change deck" with a source code, thereby upgrading the source code without making changes directly to the master file (`MERGE`, §4);

4. tidying up FORTRAN source code including relabelling targets, indentation, renumbering continuation characters, alphabetising modules, *etc.* (`TARGET`, §5);

5. comparing two ASCII files and reporting the differences found (`COMPARE`, §6), overlooking some differences of specified type;

6. splitting a long file into subroutine modules (`SPLIT`, §7); and

7. concatenating all files with a common suffix found in the current directory and all its subdirectories into a single file (`CONCAT`, §8).

These jobs are all described in the sections indicated. In addition to describing *EDITOR*'s most important task (precompilation), §2 introduces the first-time user to language of *EDITOR*. Finally, §9 describes how *EDITOR* may be installed on a new platform.

2 `PRECOM`: Precompiling source code

2.1 Basic precompiling

The main purpose of *EDITOR* is to precompile large FORTRAN source codes. By default, the *EDITOR* precompiler creates a separate file (whose name is the same as the original file with the extension `.f` appended) containing the precompiled source listing. The file containing the original source code is left as is. *EDITOR* was designed to mimic *HISTORIAN*, the precompiler available under the Cray Time Sharing System (CTSS) which was widely used on Cray machines before 1989 and, in the opinion of the author, one of the most useful and flexible precompilers of its day. In the years since, *EDITOR* has undergone many changes that has taken it beyond *HISTORIAN* and it remains the precompiler for the *ZEUS* family of astrophysical MHD codes (`www.ica.smu.ca/zeus3d`).

To use *EDITOR*, one must insert various types of *EDITOR* commands, all relatively unobtrusive, into an existing FORTRAN source code. All *EDITOR* commands go on separate lines and begin with an asterisk (`*`) in the first column. There may be one *EDITOR* command per line. Depending on the task chosen, *EDITOR* will make from 1 to 7 passes through the source code carrying out various directives as specified by the *EDITOR* commands. During precompilation, the resulting source code will be standard FORTRAN, void of any *EDITOR* commands and ready for the compiler.

It is unlikely that users with small, easily managed source codes will want to bother with any precompiler. But curators of particularly large codes which offer a variety of features, run under various operating systems (OS), and modified by several people simultaneously will want to consider some sort of preprocessor such as *EDITOR*. For example, codes which need to operate under more than one OS will almost certainly require a separate version for each OS to accommodate the differences among the host machines. The last thing that a curator of a large code wants to do is to have multiple versions of the same code to upgrade every time there is a change. The precompiler in *EDITOR* will allow these disparate versions to be merged into a single master code, and thus upgrades need be implemented only once.

EDITOR considers a source code as being made up of separate "decks" (a throw-back from the days when computer programs consisted of decks of cards), which may or may not be grouped into designated "groups". There are two types of decks that *EDITOR* recognises. "Ordinary decks" normally consist of individual program modules, such as subroutines, functions, and the main program. "Common decks" are pieces of code which are to appear *verbatim* in one or more ordinary deck(s). Common decks can be thought of as *EDITOR*'s answer to `include` statements which is a common though not ANSI-standard extension of many FORTRAN compilers. Normally, common decks consist of common block definitions which are required by more than one program module. Common decks could also be used as a way of "in-lining" a segment of code into more than one place throughout the master code.

The first thing a user should do in preparing a source code for *EDITOR* is to insert `*deck` and `*cdeck` statements at the beginning of all source code modules. The syntax is as follows:

`*cdeck` *deckname*
`*cd` *deckname*

`*deck` *deckname*
`*dk` *deckname*

where *deckname* is a user-designated name for the deck unique from all other decknames. `*cdeck` (or equivalently, `*cd` for short) tells *EDITOR* that everything that follows up to but not including the next `*cdeck`, `*cd`, `*deck`, or `*dk` statement belongs to the common deck so named. Similarly, `*deck` (`*dk`) indicates an ordinary deck. One is free, for example, to give an ordinary deck the same name as the module (*i.e.* program, subroutine, or function name) it contains.

Optionally, the user may designate "groups" of decks with the `*group` (or `*gp` for short) statement.

`*group` *groupname*
`*gp` *groupname*

where *groupname* is a user-designated name for the group, not necessarily unique from those named in other group statements. All ordinary decks will then be considered part of the group named in the most recent group statement. The `*group` command is designed for user convenience; it has no effect on how the program is compiled. They allow, for example, the FORTRAN tidy-up routine (`TARGET`, see §5) to re-alphabetise all decks within a certain group, then arrange the groups themselves alphabetically. In some sense, one can think of groups as analogous to "directories" aiding the user to locate a particular module within a large source code.

Having designated the decks (and perhaps groups), one may now insert the various precompiler commands which will be carried out during the first two passes the precompiler makes through the code. The first pass establishes which decks the user has defined, their extent, and which *EDITOR* "macros" have been set. The second pass precompiles the source code according to the macro settings. In this way, the compiler will only see that portion of the code which the user has deemed relevant to the problem at hand.

There are two types of *EDITOR* macros – "definitions" and "aliases". Setting macros is done by inserting any number of the following *EDITOR* commands *anywhere* in the master source file, or in the change deck if `MERGE` is being used in tandem with `PRECOM` (§4). Note that these statements are *global* in that they will have effect throughout the code no matter where in the code they appear. There is, for example, no way to impose a macro for part of the source code, then "turn it off" for the rest.

`*define` *def1*, *def2*, ...
`*def` *def1*, *def2*, ...
`*alias` *alias1 alias2*
`*al` *alias1 alias2*

where *def1*, *def2*, *etc.* are user-selected alpha-numeric keywords which determine "active" segments of the source code. The `*alias` statement instructs *EDITOR* to replace all occurrences of the alpha-numeric keyword *alias1* with *alias2* (except for those which appear in comment statements). Note that `*define` and `*def` are synonyms, as are `*alias` and `*al`.

Having determined which *EDITOR* macros have been set, the precompiler makes a second pass through the source code to look for `*if define` or `*if alias` statements. These determine which segments of the program are to be included in the precompiled version of the source code which is ultimately sent to the compiler. The following lists the legal syntax for *EDITOR* `*if` statements.

1. `*if define,`*macro* – the following source code is kept provided the macro is defined by a `*define` statement somewhere in the file.

   Note that the comma following "`*if define,`" is optional. It was introduced in order to mimic *HISTORIAN* where it is *not* optional. Note to *HISTORIAN* users: the `alias` feature has no analogue in *HISTORIAN*.

2. `*if -define,`*macro* – the following source code is kept provided the macro is *not* defined by a `*define` statement somewhere in the file.

3. `*if def,.not.`*macro* – same as 2. Note that `def` is an acceptable abbreviation for `define`.

4. `*if def,`*macro1* `.and.`*macro2* – the following source code is kept provided both macros are defined by a `*def` statement somewhere in the file.

5. `*if def,`*macro1* `.or.`*macro2* – the following source code is kept provided either macro is defined by a `*def` statement somewhere in the file.

6. `*if alias` *macro*`.eq.`*phrase* – the following source code is kept provided the alias *macro* has been set to the character string *phrase* by an `*alias` statement somewhere in the file.

7. `*if al` *macro*`.ne.`*phrase* – the following source code is kept provided the alias *macro* has *not* been set to the character string *phrase* by an `*alias` statement somewhere in the file. Note that `*al` is an acceptable abbreviation for `*alias`.

8. `*else` – the following source code is kept if the source code following the previous `*if` (and all the way to this `*else` statement) was not kept, *i.e.* if the truth value of the previous `*if` is false. Note that `*el` is an acceptable abbreviation.

9. `*endif` – closes the previous `*if`, `*else` structure. All source code following the `*endif` statement is not affected by the previous `*if` or `*else` statements. For every `*if` statement, there must be an `*endif` statement which follows. Note that `*ei` is an acceptable abbreviation.

10. `*call` *deckname* – includes the contents of the common deck *deckname* at the location of the `*call` statement. Note that `*ca` is an acceptable abbreviation for `*call`.

Finally, one may insert comment statements if desired by putting an asterisk in both columns 1 and 2. These comments will appear in the master source code where the user places them, but will not be copied over to the file which *EDITOR* prepares for the compiler.

Following is a simple example showing how these statements can be used. The function of the program is simply to return the time of day. Note that the line numbers in the first five columns are included for reference only, and are not modifications made by the *EDITOR* precompiler.

Master source file:

```
  1  **
  2  **  Select operating system.  Choices are: UNICOS, CONVEXOS, SUNOS
  3  **
  4  *define UNICOS
  5  **
  6  **  Select i/o subroutine by setting an alias for WRITE.
  7  **
  8  *alias WRITE write1
  9  *cdeck implicit
 10         implicit      none
 11  *cdeck common
 12  *if define,UNICOS
 13         character*8   tod
 14  *endif UNICOS
 15  *if define,CONVEXOS
 16         character*9   tod
 17  *endif CONVEXOS
 18  *if define,SUNOS
 19         character*24  tod
 20  *endif SUNOS
 21         common / com1 / tod
 22  *cdeck declare
 23  *call implicit
 24  *call common
 25  *deck tod
 26  c===+====1====+====2====+====3====+====4====+====5====+====6====+====7==
 27  c
 28         program tod
 29  c
 30  c  PURPOSE: This program returns the time of day on various systems.
 31  c
 32  c-----------------------------------------------------------------------
 33  c
 34  *call declare
 35  c
 36  *if define,UNICOS.or.CONVEXOS
 37         external      date
 38  *endif UNICOS.or.CONVEXOS
 39  *if define,SUNOS
 40         external      fdate
 41  *endif SUNOS
 42         external      WRITE
 43  c
 44  c-----------------------------------------------------------------------
 45  c
 46  c  Get time of day ("tod").
 47  c
 48  *if define,UNICOS.or.CONVEXOS
 49         call date ( tod )
 50  *endif UNICOS.or.CONVEXOS
 51  *if define,SUNOS
 52         call fdate ( tod )
 53  *endif SUNOS
 54  c
 55  c  Write result to CRT using desired i/o routine aliased to WRITE.
 56  c
 57         call WRITE
 58         stop
 59         end
 60  c
 61  *deck write1
 62  c===+====1====+====2====+====3====+====4====+====5====+====6====+====7==
 63  c
 64  c       subroutine write1
 65  *if alias WRITE.eq.write1
 66  c
 67  c  PURPOSE: This subroutine writes "tod" to the CRT.
 68  c
 69  c-----------------------------------------------------------------------
 70  c
 71  *call declare
 72  c
 73  c-----------------------------------------------------------------------
 74  c
 75         write(6,2000) tod
 76  *if define,UNICOS
 77  2000   format('Time of day according to the Cray is: ',a)
 78  *endif UNICOS
 79  *if define,CONVEXOS
 80  2000   format('Time of day according to the Convex is: ',a)
 81  *endif CONVEXOS
 82  *if define,SUNOS
 83  2000   format('Time of day according to the Sun is: ',a)
 84  *endif SUNOS
 85  *endif
 86         return
 87         end
 88  c
 89  *deck write2
 90  c===+====1====+====2====+====3====+====4====+====5====+====6====+====7==
 91  c
 92         subroutine write2
 93  *if alias WRITE.eq.write2
 94  c
 95  c  PURPOSE: This subroutine writes "tod" to the CRT using a different
 96  c           format than WRITE1.
 97  c
 98  c-----------------------------------------------------------------------
 99  c
100  *call declare
101  c
102  c-----------------------------------------------------------------------
103  c
104         write(6,2000) tod
105  2000   format('Time of day is: ',a)
106  *endif
107         return
108         end
109  c
```

With the *EDITOR* macros settings as defined in lines 4 and 8, the precompiler would convert this code into the following form which would be read by the compiler. Note that the line numbers from the master code have been preserved.

Precompiled source file:

```
 26  c===+====1====+====2====+====3====+====4====+====5====+====6====+====7==
 27  c
 28  c       program tod
 29  c
 30  c  PURPOSE: This program returns the time of day on various systems.
 31  c
 32  c-----------------------------------------------------------------------
 33  c
 10        implicit      none
 13        character*8   tod
 21        common / com1 / tod
 37        external      date
 42        external      write1
 43  c
 44  c-----------------------------------------------------------------------
 45  c
 56  c  Get time of day ("tod").
 47  c
 49        call date ( tod )
 54  c
 55  c  Write result to CRT using desired i/o routine aliased to WRITE.
 56  c
 57        call write1
 58        stop
 59        end
 60  c
 62  c===+====1====+====2====+====3====+====4====+====5====+====6====+====7==
 63  c
 64  c       subroutine write1
 66  c
 67  c  PURPOSE: This subroutine writes "tod" to the CRT.
 68  c
 69  c-----------------------------------------------------------------------
 70  c
 10        implicit      none
 13        character*8   tod
 21        common / com1 / tod
 72  c
 73  c-----------------------------------------------------------------------
 74  c
 75        write(6,2000) tod
 77  2000  format('Time of day according to the Cray is: ',a)
 86        return
 87        end
 88  c
 90  c===+====1====+====2====+====3====+====4====+====5====+====6====+====7==
 91  c
 92        subroutine write2
107        return
108        end
109  c
```

In this example, the *EDITOR* definitions are used to account for the differences in system calls under various operating systems (*e.g.*, lines 12–20 in the master code). They may also be used to tailor a code so that the compiler will generate a binary code optimised for the problem to be solved. For example, if the master source code is one which computes self-gravitating hydrodynamical flows and it has been determined that self-gravity is irrelevant to the problem at hand, one does not want to waste memory by declaring gravitational variables, nor does one want to perform the computations necessary to evolve the unwanted gravitational potential. *EDITOR* definitions may be used to eliminate those portions of the code peculiar to the self-gravity feature, thereby streamlining the code for a non-self-gravitating problem.

Note the two uses for alias macros illustrated in this example. Aliases can be used to select the module to which execution is passed without having to change the body of the source code itself (line 57). They can also be used to prevent an unwanted segment of the code from being compiled (line 93). While the latter function may be performed by the definition macros, the former may not and thus the alias feature represents a real extension of the functionality of *HISTORIAN*.

Common decks (such as `implicit`, `common`, and `declare` beginning on lines 9, 11, 22 respectively) appear in the precompiled version of the code only if they are inserted in the code by a `*call` statement (*e.g.*, line 34). Common decks not referred to by a `*call` statement will not appear in the precompiled version of the code and, in particular, the common decks themselves are not echoed to the precompiled version as separate modules. Common decks may call other common decks (*e.g.*, lines 23 and 24), but they may not call themselves. Calls to common decks may be nested (*i.e.*, common deck 1 calls common deck 2 which calls common deck 3, *etc.*) as many as 10 deep, so long as none of the common decks in the nest are the same (no closed loops!).

2.2 The `NAMELIST` extension

Namelists first appeared at the Lawrence Livermore Labs around 1980 and was incorporated as a CTSS FORTRAN extension. Since then, most operating systems offer FORTRAN77 compilers with namelist extensions, and namelist has become a standard feature of FORTRAN90. However, until the arrival of FORTRAN90 (sometime in 1994!), there was no namelist standard, and this caused great headaches for the *ZEUS* development project which, in 1990, had to operate on three platforms (UNICOS, CONVEXOS, and SUNOS, the precursor to SOLARIS). Thus, a portable namelist emulator was developed and *EDITOR* can be instructed to replace all namelist syntax in the code with calls to a namelist library, which includes features that the FORTRAN90 namelist does not (*e.g.*, ability to assign values to 2D arrays, allowing variables passed by a subroutine argument list to be namelist variables). The discussion in this subsection, therefore, is restricted to the *EDITOR* flavour of namelists.

Namelists provide an extremely useful and flexible way of supplying user-determined data to a program. Traditionally, FORTRAN source codes have relied upon prompting users for data and/or formatted reads to disc files. For many input parameters, the former can tax the user's patience while the latter can be frustrating because of the need to comply with a strict format for the input data. Namelists eliminate these problems, as illustrated in the following example:

```
 1        integer       in, jn
 2        parameter     ( in=100, jn=100 )
 3  c
 4        character*128 cscalar, cvector(in), carray(in,jn)
 5        integer       iscalar, ivector(in), iarray(in,jn)
 6        real          rscalar, rvector(in), rarray(in,jn)
 7        logical       lscalar, lvector(in), larray(in,jn)
 8  c
 9        ...
10        ...
11  c
12  c  Open the ASCII file "infile" which contains the namelist data and
13  c  ASCII file "outfile" to which a namelist summary may be written.
14  c
15        open (51, file='indata', status='old')
16        open (52, file='outdata', status='unknown')
17  c
18  c  Define list of parameters which may be set by namelist "data1".
19  c
20        namelist / data1 / cscalar, iscalar, rscalar, lscalar
21                         , cvector, ivector, rvector, lvector
22                         , carray , iarray , rarray , larray
23  c
24  c  Default values for namelist parameters
25  c
26        cscalar = ' '
27        iscalar = 0
28        rscalar = 0.0e00
29        lscalar = .false.
30        do 10 i=1,in
31          cvector(i) = ' '
32          ivector(i) = 0
33          rvector(i) = 0.0e00
34          lvector(i) = .false.
35  10    continue
36        do 30 j=1,jn
37          do 20 i=1,in
38            carray(i) = ' '
39            iarray(i) = 0
40            rarray(i) = 0.0e00
41            larray(i) = .false.
42  20      continue
43  30    continue
44  c
45  c  Read namelist from logical unit 51.
46  c
47        read ( 51, data1 )
48  c
49  c  Write a namelist summary to logical unit 52.
50  c
51        write ( 52, data1 )
```

Namelists read data from an ASCII namelist input data file (named `indata` in this example) which the user prepares before executing the binary. Line 15 is an ordinary `open` statement appropriate for linking an existing ASCII disc file to the program at execution time. Unit 51 was chosen arbitrarily. Most any unit number other than 5 or 6 may be chosen. The namelist statement on line 20 defines which variables belong to the namelist `data1` and thus which variables may be assigned values in the namelist input data file `indata`. Line 47 is where the data in the namelist `data1` are read from `indata`. As many namelists as desired may be so defined and read throughout the program and for every namelist defined, there should be a corresponding entry in the namelist input data file `indata`, as described below. Line 51 writes a summary of the namelist settings to unit 52. This is not a necessary part of the namelist structure, and many users may not want to bother with this feature. Indeed, many implementations of namelist don't even offer the option to write a summary of the namelist settings. The *EDITOR* namelist does, however, support the `write` (or equivalently, `print`) statement.

While it is not necessary, it may be useful to assign all the namelist parameters default values before the `read` statement (lines 26 through 43). This is so that in case the user does not assign values to all the parameters in the namelist input data file (by no means a requirement), all parameters will be initialised to *something* and hopefully to a useful default value.

To use the *EDITOR* namelist, one must abide by the following rules:

1. Do not use `namelist` as a variable name in any module that reads data from namelists.

2. Do not name any subroutines `nlsdac`*nn*, where *nn* varies from `01` to `24`.

3. Link the library `namelist.a` for single precision applications, and `dnamelist.a` for double precision applications to your code during the link step. The `namelist.f` and `dnamelist.f` source codes and the script files to build the libraries are in the directory `nmlst` of `dzeus35.tar` downloaded from `www.ica.smu.ca/zeus3d`).

Otherwise, use namelists as indicated above, and prepare your namelist input data file as in the following example:

```
1  c ==+====1====+====2====+====3====+====4====+====5====+====6====+====7==
2   $data1    rscalar=1.0, iscalar=1, lscalar=.true., cscalar='abcdefg'
3           , ivector=3*1,2,3, ivector(20:30)=2, ivector(40)=4*3
4  c        , ivector(1)=1,1,1,2,3, ivector(20)=11*2, ivector(40)=3,3,3,3
5           , iarray(1:100,1:100)=1, iarray(1:10,91:100)=2
6           , cvector(3)="abcdefghijklmnopqrstuvwxyz01234567890ABCDEFGHIJKL
7    MNOPQRSTUVWXYZ", lvector(98)=.t.,.false.,.f.                        $
```

Various aspects of this data file warrant discussion. Again, line numbers are *not* part of the data file; they are included for reference only.

1. Each line in a namelist input data file may be at most 72 characters wide. Anything beyond the 72nd column will be ignored. Thus, one might consider including a number line, such as line 1, which will allow one to see at a glance whether the data extend beyond the 72nd column.

2. Only one of three characters are allowed in column 1: `c`, `C` (for "commenting out" lines as in lines 1 and 4), or a blank. Anything else will cause an error message to be generated and execution to abort. Note that all lines commented out will be echoed to the CRT at execution time.

3. Only one of two characters are allowed in column 2: the `$` sentinel (for opening a namelist), or a blank. Anything else will cause an error message to be generated and execution to abort.

4. The first word to appear after the opening `$` sentinel should be the namelist name as defined in the program—in this case, `data1`. There may be spaces between the opening `$` sentinel and the namelist name, but nothing else.

5. After the namelist name, the user is free to set whichever variables within the namelist to whatever values are desired. Note that the order in which the variables are set need not be the same as the order in which they are listed in the namelist statement. One does not need to set all the variables listed either. However, any variable set in the data file which is not listed in the corresponding namelist statement in the program will generate an error message and abort execution.

6. Variable assignments may appear anywhere between and including columns 3 and 72. Variable assignments are separated by a comma and as many (including none) blanks as desired.

7. Variables are set by typing the variable name, followed by an equals sign (`=`), followed by the desired value in a format consistent with the variable type.

8. Legal values which may be assigned to a logical variable are `.true.`, `.t.`, `.false.`, or `.f.`, where the first two and last two are synonyms. Note that the periods must be included.

9. Character strings are set by enclosing the desired text inside a pair of single quotes or a pair of double quotes. If a character string is too long to fit within the 72 column constraint, one may type all the way to the 72nd column and resume the string in the third column on the following line. Note that even with the character string so split, only one opening and one closing quote should be used (*e.g.*, lines 6 and 7). This "wrap-around" feature is supported by the *EDITOR* namelist for character variable assignments only.

10. Hollerith strings are *not* supported by the *EDITOR* namelist. Use true character variables.

11. Setting values for vectors may be done in a number of ways, as illustrated on lines 3 and 4. Thus `ivector=3*1,2,3` will set `ivector(1)=1`, `ivector(2)=1`, `ivector(3)=1`, `ivector(4)=2`, and `ivector(5)=3`; `ivector(20:30)=2` will set inclusively the 20th through 30th elements of `ivector` to 2; and `ivector(40)=4*3` will set the 40th through 43rd elements of `ivector` to 3. Note that line 4 which is commented out would perform the identical assignments. Note that `ivector=3*1,2,3` is, by convention, identical to `ivector(1)=3*1,2,3`. The redundancy in notation for assigning vector values is so that namelist input data files prepared for the CTSS namelist may be read by the *EDITOR* namelist.

12. Setting values for rank 2 arrays may be done only by using the full colon notation (line 5). This notation is peculiar to the *EDITOR* namelist and, so far as is known by the author, is not supported by FORTRAN90 namelist. Setting values for arrays of greater rank than 2 is not supported.

13. The last character on the last line of a namelist assignment must be the closing `$` sentinel. Be careful that this does not go beyond the 72nd column. If it is left out or inadvertently placed beyond the 72nd column, an error message will be generated, and execution will abort.

14. One is free to define as many namelists as desired in the source code. However, once one namelist is defined, it must be read from the namelist input data file by a `read` statement (line 47 above) before a new namelist is defined. Thus, only one namelist may be pending at a time. This is not a restriction of FORTRAN90 namelist.

15. The order of the namelists (not the variables, but the namelists themselves) must be the same in the namelist input data file as they are read by the source code. Thus, if the source code is written so that namelist `data2` is read after namelist `data1`, then the variable assignments for `data2` must appear *after* the variable assignments for `data1` in the namelist input data file. If `data2` should appear before `data1` in the namelist input data file, the read to `data1` will cause the data for `data2` to be skipped. Thus, when it comes time to read `data2`, these data will not be found and an error message will ensue.

16. For every namelist read by the source code, there must be an entry with the same namelist name in the namelist input data file. If, for example, none of the parameters for namelist `data2` are to be assigned values, it is still necessary to include a minimal entry in the namelist input data file of the form:

    ```
    $data2 $
    ```

    Failure to do so will generate an error message and abort execution.

Should any of these rules be broken, namelist error messages are generated at run time. These are discussed in §A.4.

2.3 Inserting micro-tasking directives

While single processor speed is still increasing with each new technology released, most of the progress in raw compute power over the past decade has been through parallelisation. Two paradigms for parallel computing have emerged. The "Beowulf" is an example of a *distributed* multi-processing environment in which memory is distributed over the constituent processors. A great deal of thought must be given to how the processors communicate with each other, and for many applications this can be a very time-consuming task. *Shared* multi-processing (SMP) is the second and more expensive parallel environment, but can simplify enormously the task of parallelising a code. On an SMP, all processors have access to the same memory and communication among the processors is greatly reduced. The "auto-parallelisation" feature of *EDITOR* is designed for applications on SMP architectures.

There are two basic strategies to "multi-tasking" (*i.e.*, parallelising) a source code. One may "macro-task" a code by arranging for the individual processors to work on individual calls to one or more subroutine(s) whose results are independent of each other, and/or one may "micro-task" a code by sending separate iterations through a do-loop to separate processors. Of the two micro-tasking is really the only way to fully exploit the inherent parallelism in a code.

Critical to micro-tasking a code is the concept of "private" and "shared" variables. Private variables are those for which each processor has a separate and independent copy. Shared variables are those which all processors read from and perhaps write to. The rules for determining which variables are private and which are shared are fairly straight forward. A variable is private if:

1. The first time it appears within a do-loop structure, it appears on the left hand side of an equals sign; and

2. It is not indexed by the outer do-loop index.

Otherwise the variable is shared. The key to micro-tasking is to identify correctly which variables are private and which are shared—a task known as "scoping". Only if the variables are scoped properly can a micro-tasked code be generated which yields the same results as the original serial code. A number of platforms offer compiler options to help users macro-task and micro-task their codes. In developing the *ZEUS* code, the author found that the auto-tasking features provided by Cray around 1990 often made scoping errors, and/or were too timid in some of the loops they attempted to scope and a more aggressive and accurate auto-scoping feature was incorporated into *EDITOR*.

After scoping the variables in a nested loop, *EDITOR* inserts the appropriate "`cmic$`" auto-tasking directives recognised by UNICOS and SUNOS FORTRAN compilers or "`c$omp`" for `OpenMP` at the beginning of the scoped loop. Should the code then be passed through the compiler's auto-tasker, the presence of these `cmic$` or `c$omp` statements will tell the auto-tasker that these loops have already been scoped and parallelised, and can be passed over.

For all other parallelisation opportunities, the vendor's auto-tasking tools should be used. Indeed, the auto-scoping features of most vendors compilers may by now be superior to *EDITOR*'s and it may be better to use the vendor's auto-parallelisation features exclusively.

Note that the *EDITOR* auto-scoper will only scope those variables declared at the beginning of the program module. Undeclared scalars (allowed when `implicit none` is *not* used) will *probably* be scoped correctly by the compiler's auto-tasker by virtue of the `autoscope` (`default(__auto)`) directive included in all *EDITOR*-supplied `cmic$` (`c$omp`) statements but, in the author's experience, not necessarily.

Below are some examples using Cray's `cmic$` syntax to illustrate the use of the *EDITOR* auto-scoper. There is a one-to-one correspondence between all `cmic$` and `c$omp` commands which can be gleaned from the code (`edit21`) or from any `OpenMP` manual if the reader is interested. In all cases, assume that all variables including scalars have been specifically declared as real, integer, *etc.* at the beginning of the program module. Note that the line numbers in the first 5 columns are included for reference and are not modifications made by the *EDITOR* precompiler to the source code.

Example 1: A straight forward nested loop. *EDITOR* has scoped the nested loop and has inserted the appropriate `cmic$` directives. Only the outer loop is micro-tasked. On a vector machine such as a Cray, the inner loop will be vectorised.

```
  1  cmic$ do all private ( k, kp1, j, jp1, i, ip1, b1av, b2av, b3av )
  2  cmic$1       shared  ( kmax, jmax, imax, b1, b2, b3, btot )
  3  cmic$1       autoscope
  4         do 30 k=1,kmax
  5           kp1 = k + 1
  6           do 20 j=1,jmax
  7             jp1 = j + 1
  8             do 10 i=1,imax
  9               ip1         = i + 1
 10               b1av        = b1(i,j,k) + b1(ip1,j  ,k  )
 11               b2av        = b2(i,j,k) + b2(i  ,jp1,k  )
 12               b3av        = b3(i,j,k) + b3(i  ,j  ,kp1)
 13               btot(i,j,k) = 0.25 * ( b1av**2 + b2av**2 + b3av**2 )
 14               btot(i,j,k) = amax1 ( sqrt(btot(i,j,k)), tiny )
 15  10         continue
 16  20       continue
 17  30     continue
```

The variable `tiny` (line 14) was not scoped (*i.e.*, *EDITOR* did not include it in either the private or the shared lists in lines 1 and 2) because in the program from which this example was extracted, `tiny` is a *parameter* and not a *variable*. Parameters are not scoped since the compiler replaces parameters with their assigned numerical values.

Functions and subroutines should *not* be scoped. Note that the *EDITOR* auto-scoper is able to distinguish between proper arrays with argument lists (such as `btot` in lines 13 and 14) which are declared and scoped, and intrinsic functions (such as `sqrt` in line 14) which are not declared, nor scoped. Similarly, user-written functions whose attribute (real, integer, *etc.*) is not declared will not be scoped. User-written functions whose attribute is declared will not be scoped *provided they are also declared as external*. A user-written function with declared attribute not declared external will be scoped, and this may or may not have deleterious effects.

Example 2: If dependencies and/or reductions are found in the loop, the loop, as written, is not parallelisable and is not scoped. A reduction is when the value assigned to a private variable depends on the value of that variable as determined by a previous iteration, as is the case for `imax` below, or on another element of that variable should the variable be an array. A dependency is where a variable is first used as a shared variable, then as a private variable, as is the case for `ival` below. A loop containing either a dependency or a reduction will, in general, generate different answers depending upon whether it is run serially or in parallel and thus is non-parallelisable. Most compilers are sophisticated enough to rewrite code to eliminate most reductions and some dependencies and thus if *EDITOR* finds a non-parallel loop, it inserts a “No cmic$-directive report” below the loop (in comments) and adds no `cmic$` statement before it so that the compiler can have to have a crack at it if its auto-parallelisation feature is enabled.

In addition, I/O and character operations within a loop will prevent parallelisation.

```
       imax = 0
       ival = i1
       do 40 j=j1,j2
         do 30 i=i1,i2
           iarray(i,j) = ival
           ival = i + 1
           imax = max0 ( imax, iarray(i,j) )
30       continue
40     continue
c***********************************************************************
c*********** EDITOR NO-CMIC$-DIRECTIVE REPORT FOR LOOP 40    ***********
c***********************************************************************
c** No parallel directives issued because the following variable(s)   **
c** was/were found to generate dependencies or reductions:            **
c**                 ival           (dependency)                       **
c**                 imax           (reduction)                        **
c***********************************************************************
       do 130 j=j1,j2
         do 120 i=i1,i2
           chqty(i,j) = char ( iqty(i,j) )
120      continue
         write (iodmp) ( chqty(i,j), i=i1,i2 )
130    continue
c***********************************************************************
c*********** EDITOR NO-CMIC$-DIRECTIVE REPORT FOR LOOP 130   ***********
c***********************************************************************
c** Char. operations prevented parallel directives from being issued. **
c** I/O prevented parallel directives from being issued.              **
c***********************************************************************
```

Example 3: Occasionally, the *EDITOR* autotasker needs some help auto-scoping a nested loop. On line 12 in the example below, the variable `ar` is indexed by `mm` rather than the outer loop index `ny`, and is assigned a value which depends on some other element of `ar`. Technically, this constitutes a reduction and thus the loop is not parallelisable by *EDITOR*. However, careful examination of the index `mm` shows that for every value of `ny`, there is a unique value of `mm` and thus `mm` is really a “ghost” of `ny`. Hence, `ar` should actually be scoped as a shared variable and a reduction on a shared variable does not inhibit parallelism. *EDITOR* is not sophisticated enough to realise the link between `ny` and `mm`, and thus determines that this loop is not parallelisable by virtue of the reduction.

Of course, the user would know that in fact, `ar` should be treated as a shared variable, and this information can be conveyed to *EDITOR* by inserting a dummy statement, such as line 6 in the example below. Dummy statements have a `c**` in the first three columns and are scoped by the *EDITOR* auto-tasker. However, they are treated as inert comments by the compiler. The dummy statement indexes `ar` with `ny`, and thus `ar` is scoped as shared. But this introduces another problem: If a variable is first scoped as shared, then used later in the loop as a private variable (line 12), this constitutes, technically, a dependency! It is a bogus dependency to be sure, but *EDITOR* doesn't know that. Thus, the additional fix on line 1 is required. The directive `c*ipdepipred` instructs the *EDITOR* auto-tasker to ignore any parallel dependencies and any parallel reductions it may encounter in the nested do-loop structure immediately following the directive and auto-scope it anyway. Note that *EDITOR* auto-scoping directives differ from regular *EDITOR* commands in that they start with a `c*` rather than simply `*`. Other such directives include `c*ipdep` (ignore parallel dependencies only) and `c*ipred` (ignore parallel reductions only). Inserting both the dummy statement and the `c*ipdepipred` directive allows the *EDITOR* auto-scoper to scope the loop and insert the correct `cmic$` statements just before the beginning of the outer loop (and thus right after the `c*ipdepipred` directive).

```
c*ipdepipred
cmic$ do all private ( ny, m1, n1, nx, mm, nn )
cmic$1       shared  ( nyz, ar, m0, nxz, n0 )
cmic$1       autoscope
       do 40 ny=1,nyz
c**      ar(ny) = 0.0
         m1     = m0 + 2 * ( ny - 1 ) * nxz
         n1     = n0 + 2 * ( ny - 1 ) * nxz
         do 30 nx=1,nxz
           mm     = m1 + nx
           nn     = n1 - nx + 1
           ar(mm) = ar(nn)
30       continue
40     continue
```

Example 4: There are two situations known in which the *EDITOR* auto-scoper will incorrectly declare a loop safe for micro-tasking. The first case is where a `goto` statement appears within the nested do-loop structure. If a `goto` statement redirects execution to somewhere else within the loop, parallelism is not affected and multi-tasking is desirable. However, if execution is taken outside the nested loop structure, parallelism is destroyed, and the loop should not be multi-tasked. The *EDITOR* auto-scoper has not been endowed with the ability to distinguish where execution is redirected, and so blindly scopes the loop. In this way, the *EDITOR* auto-scoper is perhaps overly aggressive. Thus, the user should be aware of nested loops with `goto` statements which redirect execution outside the loop and instruct the *EDITOR* auto-scoper to pass over the loop. This is done with the `c*nopar` directive.

```
c*nopar
       do 30 i=i1,i2
         do 20 k=k1,k2
           do 10 j=j1,j2
             if (d(i,j,k) .gt. factor*d(ism1,j,k)) go to 40
10         continue
20       continue
30     continue
40     continue
```

Note that the `c*nopar` is interpreted only by *EDITOR*. Thus, if source code unscoped by the *EDITOR* auto-scoper is passed through the compiler's auto-tasker, an attempt may still be made by the compiler to generate the appropriate `cmic$` directives.

Example 5: The second case where the *EDITOR* auto-scoper breaks down is more subtle. The *EDITOR* auto-scoper will scope nested loops in which there is a call to a subroutine.

If the user determines that the subroutine call destroys parallelism, the loop should not be scoped, and the user should place a `c*nopar` directive before the outer loop. Now, assuming that the subroutine call does not destroy parallelism, one still has to be careful. There may still be the problem of scoping the variables in the subroutine argument list. In the example below, the variable `vp` is assigned values by the subroutine call and not by an assignment statement explicitly in the loop itself. Thus `vp` is intrinsically a private variable. However, without the dummy statement (line 7), the first appearance of `vp` is *not* to the left of an equals sign, and thus it would be scoped erroneously as shared (*EDITOR* is not sophisticated enough to search for the subroutine and scope the variables in the calling list based on the contents of the subroutine). Thus, dummy line 7 forces `vp` to be scoped as private.

There is still a potential trap, however. Normally, local variables within a subroutine will be considered private for the purpose of multitasking loops that contain calls to that subroutine. This is as it should be. However, the same local variables will be treated as *shared* if they are declared to be in common or equivalenced to variables in common by the subroutine called in the loop. This will invariably yield incorrect and non-repeatable results. Note that in the example below, `vtmp` and/or `vp` may be part of a common block. Since the `cmic$` directives explicitly tell the compiler that these variables are private, extra copies of the variables will be made regardless of whether they are in common or not. But if there is no way to tell the compiler that common variables in a subroutine called within a loop are to be treated as private, this will create irreproducible results. Unless you know such constructs will not cause problems, it is best to avoid calls to such routines inside a loop you wish to micro-task, or avoid micro-tasking the loop altogether.

```
 1  cmic$ do all private ( j, i, vtmp, vp )
 2  cmic$1       shared  ( j1, j2, i1, i2, v2, vg2, v2star, qty )
 3  cmic$1       autoscope
 4         do 60 j=j1,j2
 5           do 20 i=i1,i2
 6             vtmp(i) = v2(i,j) - vg2(j)
 7  c**        vp  (i) = 0.0
 8  20       continue
 9           call x3zc1d ( vtmp, vp )
10           do 30 i=i1,i2
11             v2star(i,j) = vp(i) * qty(i)
12  30       continue
13  60     continue
```

2.4 Splitting a source code; generating a makefile

For code developers who prefer to work with a single master file containing all the program modules, UNIX often poses a dilemma. Two useful UNIX facilities, `MAKE` and `DBX`, work best if the source code is split into individual files, one for each program module. Using `FSPLIT` is unsatisfactory because it pays no attention to which modules have been changed and which have not, forcing `MAKE` to recompile *all* the program modules, not just the ones that were changed.

The *EDITOR* precompiler offers an easy way around this problem. One may instruct the precompiler to make yet another pass through the master source file and this time split it into individual files for each module. The naming convention for these files is as one might hope—the name of the module with a specified extension (default is `.f`). One can even specify in which directory these files should be placed. Before writing a file to disc, the precompiler will check to see if there is already a file by that name on disc in the specified directory. If there is not, a new file is created. If there is, the precompiler compares line for line the version of the module it just split off the master file with the disc file. If the two differ, the disc file is updated. If the two are identical, the disc file is *not* updated. In this way, `MAKE` will not recompile unaltered program modules. For large source codes and slow compilers, this is no small consideration.

At the same time, the *EDITOR* preprocessor will generate a makefile if `makename` is specified (see §2.5). The user may tell *EDITOR* which compiler, compiler options, loader, and loader options to use in the makefile. If you need to compile a few routines with different compiler options than the majority, you may specify these special compiler options as well as the routines for which these special options apply. In addition, the desired name for the binary executable may be specified. Thus, once the *EDITOR* preprocessor has processed the master source code, the code may be compiled simply by typing:

`make -f` *makename*

where *makename* is the name of the makefile specified by the user.

2.5 The `precom.s` script file

A precompiled version of a FORTRAN source code may be generated by issuing the following command:

```
csh -v precom.s
```

where `precom.s` is an ordinary C-shell script file as follows:

```
 1  #============== SOURCE FILE TO PRECOMPILE A SOURCE CODE ===============#
 2  #
 3  #========================================> Get files from home directory.
 4  if(! -e xedit21) cp  USERID/editor/xedit21 .
 5  #----------------------> If necessary, create the directory "DIRECTORY".
 6  if(! -e DIRECTORY) mkdir DIRECTORY
 7  #----------------------> Create the input deck for EDITOR, and execute.
 8  rm -f inedit
 9  cat << EOF > inedit
10   \$editpar   inname='SOURCECODE'
11            , ibanner=1, job=4, idump=1, inmlst=1, iutask=1, safety=0.4
12            , iupdate=1, ext='.f', branch='DIRECTORY'
13            , makename='MAKEFILE', xeq='EXECUTABLE'
14  c         , coptions='-g -C -ftrap=common', loptions='-g'
15            , coptions='-fast', loptions='-fast'
16            , libs='namelist.a'                                      \$
17  EOF
18  chmod 755 xedit21
19  ./xedit21
```

A softcopy of `precom.s` may be found in the `editor` directory of `dzeus35.tar` downloaded from `www.ica.smu.ca/zeus3d`. Note that the line numbers in the first five columns are not part of the file and are included only for reference.

Comments for C-shell script files are indicated by a `#` in column 1. For all *EDITOR* C-shell script files listed in this manual, there are, by convention, two types of comments. Those lead by a double line (`==========>`) indicate that the following portion of the script file should rarely, if ever, require changing. Those lead by a single line (`---------->`) indicate segments of the script file which will probably have to be altered every time the script file is used.

The first segment of `precom.s` copies the necessary files (in this case, just the edit21 executable) to the present working directory. Note that the UNIX phrase "`if(! -e ...)`" ensures that the named file will *not* be retrieved if it already exists in the pwd.

The second segment creates a directory on disc into which all source files split from the master code during precompilation (§2.4) and all corresponding object and listing files are placed, should this option be used.

The third segment is where the input parameters for *EDITOR* are specified. Input parameters are read in by a "namelist", as discussed in §2.2. Specifying the parameters in the namelist `editpar` is how *EDITOR* is controlled. In the example, both `SOURCECODE` and `DIRECTORY` (as well as all words in allcaps) have to be specified by the user. Note that the `$` sentinel is preceded by a backslash (\). This prevents the script file from interpreting the `$` as a control character and instead treats it as an ASCII character to be passed (without the leading backslash) to the text file `inedit`.

There are 45 valid namelist parameters in `editpar`, but only 20 are relevant for precompiling source code. These include six general parameters which all or most *EDITOR* jobs use, and 14 additional parameters peculiar to `PRECOM`.

```
parameter                      description                         default

  GENERAL

  inname    name of source code to be edited (character*64).
  ibanner   =1 => print banner to screen at beginning of execution     0
            =0 => no banner
  job       =1 => number lines in source code                          1
            =2 => tidy source code (retarget, indent, alphabetise)
            =3 => update source code with change decks
            =4 => prepare source code for compilation.
            =5 => compares two files and reports location of first
                  "ndiff" differences.
            =6 => splits master file into module files.
            =7 => concatenates module files to master file.
  idump     =1 => diagnostic dumps are written to "output".            0
            =0 => no diagnostic dumps.
  outname   name of outfile.  If unspecified, outfile will be named
            according to internal naming convention (character*64).
  safety    Memory management parameter.  k2*safety lines of         0.0
            "inname" are read at a time.  0.0 => 0.9 (job=1,2,5,6,7),
            0.4 (job=3,4)

  PRECOM (job=4)

  inmlst    =0 => leave NAMELIST and associated I/O alone.             0
            >0 => substitute all occurrences of NAMELIST and
                  associated I/O for calls to subroutines in
                  (D)NAMELIST.A.  A maximum of "inmlst" assignments
```

```
                can be made in the input deck per NAMELIST.
                "inmlst"=1 => 1000.
iutask    =0 => no microtasking attempted                           0
          =1 => insert Cray microtasking directives (cmic$)
                in front of parallelisable nested do-loops.
          =2 => insert OpenMP parallelisation directives (c$omp)
                in front of parallelisable nested do-loops.
iupdate   =0 => PRECOMpiled source code is dumped to one file.      0
          =1 => PRECOMpiled subroutines stored to separate files.
                Subroutine is written only if it is different from
                version on disc, or if it doesn't exist on disc.
branch    character*32 string indicating default directory in     blank
          which to write subroutine files.
ext       desired extension for files (character*8)                 '.f'
makename  name of makefile to be created (character*16).  If      blank
          blank, no makefile is created.  The files compiled by
          MAKE are those in the directory 'branch'.
compiler  specifies compiler to be used by MAKE.         UNICOS 'f90'
          (character*32)                               CONVEXOS 'fc -c'
                                             SUNOS, AIX, LINUX 'f77 -c'
coptions  specifies compiler options.  For f90 (UNICOS),          blank
          these are appended to the string '-b \$*.o', which is
          needed for the makefile.  Similar strings for SUNOS,
          CONVEXOS, LINUX, and AIX (character*128).
speccopt  specifies compiler options for special decks named in   blank
          array specdk.  Occasionally, one needs to compile a few
          routines with special compiler options in order for the
          computations to be done correctly (character*128).
specdk    those decks to be compiled with compiler options        blank
          speccopt (character*16(k4)).
loader    specifies loader to be used by MAKE.           UNICOS 'segldr'
          (character*32)                               CONVEXOS     'fc'
                                             SUNOS, AIX, LINUX    'f77'
xeq       name of executable to be created by MAKE (character*64
          to allow for directory specification as well, default
          is 'inname' with the extension '.x').
loptions  specifies loader options other than -o.  These are      blank
          appended to '-o \$(EXE)', which is needed for the
          makefile (character*128).
libs      are the libraries to be linked by the loader.  As many  blank
          libraries can be specified as will fit in character*512.
```

Some notes:

1. Default name to be given the precompiled disc file (`outname`) is `inname` with the extension `.f`.

2. There are OS-dependent defaults for the compiler and loader. In the example of `precom.s` listed above, there are two possible compiler options spelled out. The first (line 14, commented out) is the setting for full diagnostics and to enable dbx for f77 on SUNOS (AIX, LINUX). Line 15 (not commented out) gives the appropriate compiler options for full optimisation on SUNOS (AIX, LINUX).

3. *EDITOR* is unable to interpret tab characters. For programmers who habitually use tabs in their source code, these should all be replaced manually with the appropriate number of blanks before attempting to preprocess it with *EDITOR*.

4. Should the *EDITOR* precompiler detect any *EDITOR* syntax errors, *EDITOR* will insert an error message immediately following the offending statement in the precompiled file (with the `.f` extension). Detection of error message will render the precompiled file unusable for compilation purposes, and may prevent additional passes through the code requested by the user (replacing namelists, for example). The user will be told that errors were detected and how to find them in the precompiled disc file. Once the errors are corrected in the master file, the user can attempt to precompile the file again. See §A for a description of the error messages generated by *EDITOR*.

3 NUMBER: Generating a numbered listing

3.1 Reformatting a file

The NUMBER feature of *EDITOR* will take as input any ordinary ASCII source code and reformat it so that each line of the source code appears with various labels. By default, a separate file (whose name is the same as the input file with the extension .n appended) is created containing the reformatted source listing. The original file is left as is.

Each line is labelled with as many as 6 labels. The first and third columns of the reformatted file is the line number since the beginning of the file, with the source code itself (72 characters wide) in the second column. The fourth column is the number of *executable* statements (*i.e.*, not including comments and continuation lines) since the beginning of the current module. The statement number takes into account the number of statements implied by each *call statement (§2.1). This is useful, for example, if compiler and/or debugger diagnostics refer to the executable statement number within a module, rather than the ASCII line number as more modern compilers and debuggers do. The fifth column is the number of lines since the beginning of the current module (see inumber below). The sixth column is the "group" name in which the current module is grouped (§2.1), while the seventh column lists the name of the current module. The user has some control over what labels are put on each line (see inumber below) and how the number of executable statements since the beginning of the current module is computed (see ixclude below). A full compliment of labels will expand an ordinary 72 column FORTRAN source listing to 132 columns, so an appropriate printer must be used to print out the reformatted listing. In addition, *EDITOR* may be instructed to place a table of contents at the beginning of the listing. To aid in locating a module rapidly, the table of contents list the modules both sequentially and alphabetically.

3.2 The number.s script file

A source listing may be reformatted with NUMBER by issuing the following command:

```
csh -v number.s
```

where number.s, is as follows:

```
#============= SOURCE FILE TO CREATE A NUMBERED LISTING ===============#
#
#=====================================> Get files from home directory.
if(! -e xedit21) cp  USERID/editor/xedit21 .
#----------------------> Create the input deck for EDITOR, and execute.
rm -f inedit
cat << EOF > inedit
 \$editpar   inname='SOURCECODE'
          , ibanner=1, job=1, inumber=3, itable=1, ixclude=1          \$
EOF
chmod 755 xedit21
./xedit21
```

A softcopy of number.s may be found in the editor directory of dzeus35.tar downloaded from www.ica.smu.ca/zeus3d.

The first segment gets `xedit21` from the user's home directory, if needed. The second segment prepares the input deck for *EDITOR* appropriate for reformatting a source listing. In addition to the general namelist parameters described in §2.5, there are three namelist parameters which may be used specifically to control `NUMBER`.

```
parameter                        description                        default

  NUMBER (job=1)

  inumber   =1 => sequential numbering of source code only.            3
            =2 => sequential, statement, and by subroutine.
            =3 => sequential, statement, and by deck.
  itable    =1 => creates a table of contents (inumber=3 only)         1
            =0 => no table of contents
  ixclude   =1 => won't label line with statement number if excluded   1
                  by *if, *else, *endif logic (inumber=3 only)
            =0 => labels all executable statements.
```

Some notes:

1. Selecting `inumber=3` specifies that line numbering in the fifth column will be done relative to `*deck` and `*cdeck` statements, rather than FORTRAN module statements such as `program`, `subroutine`, `function`, *etc.* (`inumber=2`).

2. Setting `ixclude=1` will exclude "dormant" parts of the source code (as determined by the settings of the *EDITOR* macro definitions; see §2.1) from the computation of the current statement number (fourth column).

4 `MERGE`: Merging source code

4.1 Change decks

One important feature of *EDITOR* is the ability to merge a "change deck" into an existing source code. This feature is useful from the standpoint of keeping the changes to a working version of a source code separate from the source code itself. Further, change decks are useful to code development projects in which there are several contributors. In principle, a curator of a code may gather in all change decks and let *EDITOR* merge these change decks into the current master code, thereby generating the next version. Finally, each code developer may independently and temporarily merge their own change deck into the current version of the code in order to develop and debug their changes.

Change decks consist largely of the new lines of FORTRAN that the user wishes to place into an existing code, along with any *EDITOR* precompiler statements that may be required. These code segments may either be inserted into the code at a specified location, or replace a specified part of existing code. There are four *EDITOR* commands that control a `MERGE`:

1. `*insert` *deckname.n* – inserts text immediately following the `*insert` command into the source code directly after line n in (c)deck *deckname*.

2. `*delete` *deckname.n,m* – deletes lines n through m in (c)deck *deckname*, and replaces it with the text immediately following the `*delete` command, if any. Note that m must be greater than n. If m is missing altogether, then $m = n$ will be assumed.

3. `*ident` *changedeckname* – identifies the name of the change deck to the code developer. It has no internal (to *EDITOR*) function, and is included for the sole purpose of preserving backward compatibility with *HISTORIAN*. In practise, it never needs to be used.

4. `*read` *filename* – replaces the statement with the contents of the named file. This is how more than one change deck may be merged with a source file at the same time.

The line numbers m and n are relative to the most recent `*(c)deck` statement, where the `*(c)deck` statement itself is line 1. The line numbers may be attained most easily from the fifth column of a listing of the source code reformatted by `NUMBER` (§3).

Note that `*i`, `*d`, `*id` and `*r` are valid abbreviations of `*insert`, `*delete` `*ident` and `*read` respectively. For those who don't like the fact that the *delete* command can actually be used to *replace* text, the command `*replace` (or `*rp` for short) is a recognised synonym for `*delete`. Use the two interchangeably. Together, `*delete`, `*insert`, and `*replace` commands shall be referred to as "`MERGE` edits".

EDITOR's `MERGE` may be given *one* master source file and *one* change deck (containing an arbitrary number of `*reads` to other change decks if desired) and creates a new merged file whose name, by default, is the same as the input master source file with the extension `.m` appended. The merger also generates an amalgamated change deck with all the `*read` commands, if any, carried out. This change deck has the same name as the user-specified change deck with the extension `.m` appended and is where error messages, if any, are inserted. The original files are not changed.

An example might be appropriate at this point. Following is another version of the time-of-day program used in §2, a change deck, and the result of having the two merged by *EDITOR*.

Master source file (named, for example, `tod`):

```
 1  *deck tod
 2  c===+====1====+====2====+====3====+====4====+====5====+====6====+====7==
 3  c
 4         program tod
 5  c
 6  c  PURPOSE: This program returns the time of day on various systems.
 7  c
 8  c-----------------------------------------------------------------------
 9  c
10         implicit      none
11  *if define,UNICOS
12         character*8   tod
13  *endif UNICOS
14  *if define,CONVEXOS
15         character*9   tod
16  *endif CONVEXOS
17  c
18         external      date
19  c
20  c-----------------------------------------------------------------------
21  c
22  c  Get time of day ("tod").
23  c
24         call date ( tod )
25  c
26  c  Write result to CRT.
27  c
28         write ( 6, 2000 ) tod
29  2000   format('Time of day is: ',a)
30         stop
31         end
```

Change deck (named, for example, `chgtod`):

```
 1  *define UNICOS
 2  *delete tod.29
 3  *if define,UNICOS
 4  2000   format('Time of day according to the Cray is: ',a)
 5  *endif UNICOS
 6  *if define,CONVEXOS
 7  2000   format('Time of day according to the Convex is: ',a)
 8  *endif CONVEXOS
 9  *insert tod.6
10  c           Systems include UNICOS and CONVEXOS.
```

Master source file (`tod`) merged with change deck (`chgtod`) to form new master source file (`tod.m`):

```
 1  *deck tod
 2  c===+====1====+====2====+====3====+====4====+====5====+====6====+====7==
 3  c
 4         program tod
 5  c
 6  c  PURPOSE: This program returns the time of day on various systems.
 7  c           Systems include UNICOS and CONVEXOS.                         *
 8  c
 9  c-----------------------------------------------------------------------
10  c
11         implicit      none
12  *if define,UNICOS
13         character*8   tod
14  *endif UNICOS
15  *if define,CONVEXOS
16         character*9   tod
17  *endif CONVEXOS
18  c
19         external      date
20  c
21  c-----------------------------------------------------------------------
22  c
23  c  Get time of day ("tod").
24  c
25         call date ( tod )
26  c
27  c  Write result to CRT.
28  c
29         write ( 6, 2000 ) tod
30  *if define,UNICOS                                                        *
31  2000   format('Time of day according to the Cray is: ',a)                *
32  *endif UNICOS                                                            *
33  *if define,CONVEXOS                                                      *
34  2000   format('Time of day according to the Convex is: ',a)              *
35  *endif CONVEXOS                                                          *
36         stop
37         end
38  *define UNICOS                                                           *
```

Some notes:

1. MERGE commands may *not* appear in the master file, only in the change deck.

2. Anything before the first MERGE edit will be placed at the end of the merged file (*e.g.*, line 38 of tod.m).

3. An asterisk (*) is placed in the 74th column of every line put into the merged master source file by the change deck. This asterisk is only to aid the user to see at a glance which lines are new to this version of the code. If the .m file is then passed through PRECOM (§2), NUMBER (§3), or TARGET (§5), the asterisk in column 74 is not copied to the .f, .n, or .t file respectively.

4. Note that the line numbers used in the MERGE edit statements are *always* those of the original master file. That is to say, the user does not have to worry that a MERGE edit made somewhere else in the change deck might affect the line numbering for other MERGE edits. By the same token, one may not make a MERGE edit on a MERGE edit in

the current change deck. Changes to MERGE edits should be done directly in the change deck(s).

5. Obviously, care should be taken to ensure that MERGE edits do not refer to lines in the master source that have been deleted by other MERGE edits. Such a conflict will generate a (non-fatal) error message (see §A.2).

6. All the punctuation in the MERGE commands is optional. "`*d tod 1 3`" is just as valid as "`*d tod.1,3`". The punctuation used in these examples reflects *HISTORIAN* syntax which *EDITOR* permits for the sake of compatibility.

As a source code preprocessor, *EDITOR* is probably in its most useful state when the MERGE and PRECOM features are used in tandem which is accomplished by setting the appropriate input parameter (§4.2). One can take a master file and a change deck, merge them together to produce a `.m` file, precompile the `.m` file to generate a `.f` file, replace all the namelists micro-task, it, update only those modules that were affected by whatever change you might have made to the change deck, and generate the new makefile, all with one execution of *EDITOR*. It is in this mode that the author uses *EDITOR* to manage the *ZEUS-3D* code, and therefore the mode which is probably the most debugged and robust. The script file in §4.2 is a template for using *EDITOR* in just this way.

Curators of large codes should be warned that change decks can become too cumbersome and numerous to make this strategy practical. As a first guide, one might consider permanently merging a change deck with the master code once it has grown to 25% the size of the master code. Then a new change deck may be started with references to line numbers in the new version of the master source file.

4.2 The `merge.s` script file

A merged source listing may be generated by issuing the following command:

```
csh -v merge.s
```

where `merge.s`, is as follows:

```
#======= SOURCE FILE TO MERGE A CHANGE DECK INTO A SOURCE CODE ========#
#
#========================================> Get files from home directory.
if(! -e xedit21) cp USERID/editor/xedit21 .
#=========================> If necessary, create the directory DIRECTORY.
if(! -e DIRECTORY) mkdir DIRECTORY
#---------------------------------------------------> Create change deck.
rm -f changes
cat << EOF > changes
*ident changes
*delete par.3
       parameter ( idim=100, jdim=100, kdim=100 )
*read CHANGEDECK
EOF
#------------------------> Create the input deck for EDITOR, and execute.
rm -f inedit
cat << EOF > inedit
```

```
 \$editpar   inname='SOURCECODE', chgdk='changes'
           , ibanner=1, job=3, idump=1, inum=0, ipre=1, inmlst=1
           , iutask=0, iupdate=1, ext='.f', branch='DIRECTORY'
           , makename='MAKEFILE', xeq='EXECUTABLE'
           , libs='namelist.lib'                                 \$
EOF
chmod 755 xedit21
./xedit21
```

A softcopy of `merge.s` may be found in the `editor` directory of `dzeus35.tar` downloaded from `www.ica.smu.ca/zeus3d`.

The first segment retrieved `xedit21` from the user's home directory, if needed. You could add lines here to retrieve the source code and change deck from their home directory as well. The second section generates a directory on disc just as `precom.s` did (§2.5) in anticipation that the user will want to use the precompiler in tandem with the merger. The third section generates the actual change deck that will be merged with the master file. The script file will create a disc file called `changes` *after* it has removed any such disc file which may already exist. Thus, don't run this script file *verbatim* within a directory in which there is a file called `changes` that you can't live without!

The file `changes` contains the inert `*ident` command, followed by another example of a `MERGE` edit. In this case, line 3 of a deck named `par` is being replaced with a parameter statement setting `idim`, *etc.* to 100. This illustrates a structure that the author finds very useful, and is why this specific example has been included in this otherwise general template. In this case, a program has been written with all the parameter statements placed together in a common deck called `par`. Every subroutine that requires knowledge of the parameter values then has a `*call par` statement at the beginning of the declaration list. If for every job run, a different set of parameter values is required, the easiest and most accessible place to make this change is right in the script file which merges and precompiles the source code. Thus, right in `merge.s`, one might include the most often-needed changes using the `MERGE` edit structures described in this section. Then, as indicated in this template, one could issue a `*read` command which will bring in the bulk of the changes being considered at this time which are in some user-supplied file `CHANGEDECK`.

Finally, the fourth segment creates the input deck for *EDITOR* appropriate for the merge being performed, and then executes *EDITOR*. In addition to the general namelist parameters described in §2.5, there are three namelist parameters which are peculiar to `MERGE`. However, if the reformatting feature of *EDITOR* (`NUMBER`, §3) is to be called in tandem with `MERGE`, then `inum` should be set to 1 and all the input parameters peculiar to `NUMBER` (3.2) become applicable. If the precompiler is to be called in tandem with `MERGE`, then `ipre` should be set to 1, and all the input parameters peculiar to `PRECOM` (§2.5) become applicable.

parameter	description	default
MERGE (job=3)		
chgdk	name of change deck to be merged with "inname" (character*64).	
inum	=1 => a NUMBERed file will be created. =0 => no NUMBERed file	0
ipre	=1 => a PRECOMpiled file will be created. =0 => no PRECOMpiled file	0

Some notes:

1. If, on the one hand, `MERGE` and `PRECOM` are performed together on the file `myprog`, then *EDITOR* will generate two additional source files, namely `myprog.m` and `myprog.f`. The former will be the result of the merger with all the precompiler commands, if any, remaining while the latter will be a precompiled version ready for the compiler containing nothing but FORTRAN (having had all the precompiler commands carried out and then expunged). If, on the other hand, `MERGE` is used with `ipre=0`, then only a `.m` file will be generated. Then, if `myprog.m` is passed through the precompiler using `precom.s`, the precompiled file will be named `myprog.m.f`. Note that if the *EDITOR* namelist parameters in the two scenarios have the same values, then the files `myprog.f` and `myprog.m.f` will be *identical*.

2. Should the *EDITOR* merger detect any *EDITOR* syntax errors, *EDITOR* will insert an error message immediately following the offending statement in the amalgamated change deck (`.m` extension). The user will be told that errors were detected and how to find them. Once the errors are corrected in the *original* change deck (*i.e.*, not the `.m` file where the errors were reported), the user can attempt to merge the files again. See §A.2 for a description of the non-fatal error messages generated by *EDITOR*.

5 TARGET: FORTRAN tidy-up

The `TARGET` feature of *EDITOR* was designed to rewrite a user's source code with uniform origin-target labels, continuation characters, and indentation, as well as rearranging the various decks and groups alphabetically. `TARGET` can also be instructed to replace `do-enddo` structures with targeted do-loops (but not the reverse—reflecting the author's bias!). By default, a separate file (whose name is the same as the input file with the extension `.t` appended) is created containing the tidied source listing. The original file is left as is.

`TARGET`'s primary function is to resequence numbered statements ("targets") and their corresponding "origins" to regain the order that the writer may have originally intended. Examples of origin and target statements are as follows:

```
1        do 10 i=1,imax
2        ...
3        go to 20
4        ...
5  20    continue
6        ...
7  10    continue
```

Lines 1 and 3 are origins while lines 7 and 5 are their respective targets.

`TARGET` can recognise virtually all FORTRAN structures in which origins and targets may lurk. These include:

1. (nested) do-loops with numbered targets

```
      do 100 i=1,imax
      do 100 j=1,jmax
100   ...
```

2. go to statements

```
      go to 20
20    ...
```

3. computed go to statements

```
      go to (10,20,30) i
10    ...
20    ...
30    ...
```

4. if statements

```
      if (j.eq.1) go to (10,20) i
10    ...
20    ...
```

5. computed if statements,

```
      if (10,20,30) x
10    ...
20    ...
30    ...
```

6. i/o statements (`write`, `print`, `encode`, `read`, `decode`, `open`)

```
      write (6,1000) x
1000  format(f5.2)
      read (lu,end=20,err=100) x
20    ...
100   ...
      open (unit=lu,file=infile,status='old',err=10,form='unformatted')
10    ...
```

As mentioned, `TARGET` may be instructed to convert all `do-enddo` structures into targeted do-loops. `TARGET` will scan each program module (*i.e.* deck, program, function, subroutine) for origin and targets and reassign the numerical values of the labels so that the targets (not the origins) appear sequentially (with a specified increment between each consecutive target). Three levels of targets are identified and are resequenced independently. All do-loop and `goto` targets (including the `end` and `err` options in the parameter lists of `read` and `open` statements) are considered together and by default, are assigned labels between 10 and 990. All input statements (`read`, `decode`) are resequenced between 1010 and 1990, and all output statements (`write`, `print`, `encode`) are resequenced between 2010 and 2990. This allows one to identify, at a glance, which targets belong to `do`/`goto` statements, which belong to input statements, and which belong to output statements.

As part of the resequencing process, `TARGET` will force all do-loop and `goto` statements to "land" on a `continue` statement, thus displacing the original targeted statement by one line. For do-loop targets, the `continue` statement is put *after* the original targeted statement while for `goto` statements, the `continue` statement is placed *before* the original targeted statement. This leaves the logical intent of the code intact.

Besides resequencing origin-target statements, `TARGET` may be instructed to relabel continuation statements and force uniform indentation. These three features are illustrated in the following example:

Original code:

```
      subroutine sub1 (in, jn, x, y, array, iret)
c
      real x(in), y(jn), array(in,jn)
c
      go to (1,31) iret
31    continue
      do 10 j=1,jn
      do 10 i=1,in
10    array(i,j) = x(i)**2
     .           + 2.0 * x(i) * y(j)
     .           + y(j)**2
      if (imax.le.100) then
      imax = 100
      else
      imax = 200
      endif
1     continue
      return
      end
```

Tidied code:

```
       subroutine (in, jn, x, y, array, iret)
c
       real x(in), y(jn), array(in,jn)
c
       go to (40,10) iret
10     continue
       do 30 j=1,jn
         do 20 i=1,in
           array(i,j) = x(i)**2
     1                + 2.0 * x(i) * y(j)
     2                + y(j)**2
20       continue
30     continue
       if (imax.le.100) then
         imax = 100
       else
         imax = 200
       endif
40     continue
       return
       end
```

Indentation is applied to both (nested) do-loops and if-else-endif structures as illustrated above. Note that applying uniform indentation forces the source code to begin in column 8, rather than column 7, the minimum allowed by FORTRAN syntax. Starting in column 8 means that there will always be at least one space between a continuation character in column 6 and the first character in the statement, thus improving readability. Note also that applying uniform indentation will preserve any vertical structure imposed by the user. Thus, in the example above, the `+`s remain under the `=`. If the application of uniform indentation (or resequencing origins) causes the line to extend beyond the 72nd column, `TARGET` will break the line at the 72nd column (without regard to word breaks) and generate a continuation statement with an ampersand (`&`) in column 6. This will not affect the logic of the source code, but may offend the user's notion of aesthetics. Thus, after `TARGET` has finished with the source code, one merely needs to search the tidied version for an `&` in column 6 and then make the desired changes manually.

Resequencing continuation characters will cause continuation statements to be given numerical continuation characters in the following sequence: 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, *etc.* Zero (0) is not used, since it is an illegal continuation character in FORTRAN.

Finally, `TARGET` can be instructed to rearrange the modules alphabetically according to the full deck name which consists of the group name (as determined by the most recent `*group` statement; §2.1) followed by the deck name (as determined by the most recent `*deck` or `*cdeck` statement; §2.1). Common decks will be placed before ordinary decks. The user may single out a few decks to be placed before all else regardless of where they belong alphabetically. Thus, for example, if there were a deck containing opening comments, or if it was desired to place the program before the subroutines, this may be accommodated by setting the appropriate namelist parameters, as described in the next subsection.

5.1 The target.s script file

A source listing may be tidied by issuing the following command:

```
csh -v target.s
```

where `target.s`, is as follows:

```
#============ SOURCE FILE TO TIDY UP FORTRAN SOURCE CODE ==============#
#
#=====================================> Get files from home directory.
if(! -e xedit21) cp  USERID/editor/xedit21 .
#----------------------> Create the input deck for EDITOR, and execute.
rm -f inedit
cat << EOF > inedit
 \$editpar   inname='SOURCECODE'
          , ibanner=1, job=2, idump=1
          , ibegdo=  10, ienddo= 990, ibegre=1010, iendre=1990
          , ibegwr=2010, iendwr=2990
          , inc=10, irepl=1, ireseq=1, indent=2, ialpha=1
          , first  ='ABSOLUTE FIRST  (C)DECK'
          , secnd  ='ABSOLUTE SECOND (C)DECK'
          , firstcd='BEFORE ALL CDECKS, BUT AFTER secnd'
          , firstdk='BEFORE ALL  DECKS, BUT AFTER secnd'              \$
EOF
chmod 755 xedit21
./xedit21
```

A softcopy of `target.s` may be found in the `editor` directory of `dzeus35.tar` downloaded from `www.ica.smu.ca/zeus3d`.

The first segment retrieves `xedit21` from the user's home directory, if needed. The second segment prepares the input deck for *EDITOR* appropriate for tidying up a source listing. In addition to the general namelist parameters described in §2.5, there are 15 namelist parameters peculiar to `TARGET`. These are described below.

parameter	description	default
TARGET (job=2)		
ibegdo	lowest target value for "goto" and "do" stmnts.	10
ienddo	highest target value for "goto" and "do" stmnts.	990
ibegre	lowest target value for "read" and "decode" stmnts.	1010
iendre	highest target value for "read" and "decode" stmnts.	1990
ibegwr	lowest target value for "write" and "encode" stmnts.	2010
iendwr	highest target value for "write" and "encode" stmnts.	2990
inc	increment between successive targets.	10
irepl	=1 => replace "do-enddo"s with targeted "do"s.	1
	=0 => do not replace "do-enddo"s.	
ireseq	=1 => resequence 6th character in continuation stmnts.	1
	=0 => no resequence.	
indent	=0 => no uniform indentation is imposed.	2
	>0 => impose uniform indentation of "indent" spaces.	
ialpha	=0 => no alphabetisation of decks.	1
	=1 => modulo specification of "first" etc., "common decks" and "ordinary "decks" are arranged alphabetically. "Common decks" are listed before "ordinary decks".	

```
first     name of "deck" (common or ordinary) to be listed before
          all others (character*16).
secnd     name of "deck" (common or ordinary) to be listed right
          after "first" (character*16).
firstcd   name of "common deck" to be listed before all other
          "common decks" but after "secnd" (character*16).
firstdk   name of ordinary "deck" to be listed before all other
          ordinary "decks" but after "secnd" (character*16).
```

It should be noted that `TARGET` is particularly sensitive to tab characters. *EDITOR* is known to make mistakes and even crash if it encounters a tab character during a `TARGET` session and so, as mentioned in §2.5, all tab characters should be replaced with the appropriate number of blanks.

6 COMPARE: Comparing similar ASCII files

6.1 Comparing entire files

EDITOR may be instructed to compare two ASCII files character for character and report the differences found in the two files. The locations of where the files diverge and reconverge are reported. Obviously, for two totally dissimilar files, such a report may become unwieldy as chance alignments of the two files are discovered.

What makes *EDITOR*'s `COMPARE` different from `diff` on most UNIX platforms is that it may be instructed to ignore superficial or unimportant differences such as those found in FORTRAN comments (*i.e.*, lines with a `c` or `C` in the first column), the number of blanks left between "words", and the FORTRAN continuation characters chosen (character in column 6). This is an attempt to get beyond the most common differences in programming style and uncover only those differences which may alter the logic of the program. `COMPARE` is also much better at finding where the files reconverge than `diff`. Finally, if there are an unruly number of differences, one may instruct *EDITOR* to stop searching after a specified number of differences have been found.

EDITOR reports the differences by dumping the portion of the line from both files where the difference was found and points to the very character which triggered the report. It also tells the user on which line the two files reconverge. Below is an example of two files with some differences, and the *EDITOR* `COMPARE` reports generated by comparing these two files un various ways.

File 1 (`tod1`):

```
 1       program tod
 2 c-------------> This program returns the time of day on various systems.
 3       character*8   tod
 4 c
 5       call date (tod)
 6       write (6,10) tod
 7 10    format('Time of day is: ',a)
 8       stop
 9       end
```

File 2 (`tod2`):

```
 1 c===+====1====+====2====+====3====+====4====+====5====+====6====+====7==
 2 c
 3        program tod
 4 c
 5 c  PURPOSE: This program returns the time of day on various systems.
 6 c
 7 c-----------------------------------------------------------------------
 8 c
 9        implicit      none
10        character*8   tod
11 c
12        external      date
13 c
14 c-----------------------------------------------------------------------
15 c
```

```
 16  c  Get time of day ("tod").
 17  c
 18         call date ( tod )
 19  c
 20  c  Write result to CRT.
 21  c
 22         write ( 6, 2000 ) tod
 23  2000   format('Time of day is: ',a)
 24         stop
 25         end
```

These two versions of (presumably) the same program show different programming styles. The task is to find all important differences between these two files, if any. In the report that follows, no superficial differences were ignored and `COMPARE` behaves much like `diff`.

```
COMPARE :
COMPARE : diff        "tod1                      "  "tod2                      "
COMPARE :             line          context         line          context
COMPARE : ----------------------------------------------------------------------
COMPARE :   1 diff at     1                 progra      1          c===+====1==
COMPARE :     same at     4           ^                 2           ^
COMPARE :   2 diff at     5           call date (t      3            program tod
COMPARE :
COMPARE : Number of differences reported =    2
COMPARE :
```

First the anatomy of this report. The first column (headed by `diff`) labels the differences sequentially. The second column (headed by `tod1`, the name of the first file) gives the line number and a segment of the indicated line in which the difference was discovered for the first file. The third column (headed by `tod2`, the name of the second file) gives similar information for the second file. Each difference report consists of two lines; where the files diverge (`diff at`) and where the files reconverge (`same at`). The "`diff at`" line gives the context of the file in which the difference was discovered centred over a carat (ˆ) which indicates the first character found to be different. For entirely different lines, this will generally be the first character in the line. The "`same at`" line indicates on which line in each file the files reconverge, and contains the carats.

This particular report is of little use. It basically states that the two files differ right from the start (note that one file is indented with six blanks, the other with seven), happen to be the same again where both files have an empty comment line, then differ throughout the rest of the files. Since the files never reconverge, the "`same at`" line for the second difference report is never issued, and thus neither are the carats. These two files are basically too different for an all-difference report to be of much use.

In the second report which follows, differences stemming from blanks were ignored. It is a bit more useful than the first report since reported differences are more localised.

```
COMPARE :
COMPARE : diff        "tod1                      "  "tod2                      "
COMPARE :             line          context         line          context
COMPARE : ----------------------------------------------------------------------
COMPARE :   1 diff at     1                program      1          c===+====1==
COMPARE :     same at     1           ^                 3           ^
COMPARE :   2 diff at     2          c------------      4          c
COMPARE :     same at     4          ^                 4           ^
COMPARE :   3 diff at     5               call dat     5           c  PURPOSE:
COMPARE :     same at     5          ^                18           ^
COMPARE :   4 diff at     6               write (6    19           c
COMPARE :     same at     8          ^                24           ^
COMPARE :
COMPARE : Number of differences reported =    4
COMPARE :
```

The first difference reports that the first two lines of each file are different (`tod2` begins with a comment, `tod1` begins with the program statement) and that the two files reconverge again at line 1 of `tod1` and line 3 of `tod2`. Still, there is chaff amongst the wheat if one is not interested in the differences generated by comment statements.

Finally, the third report which follows was generated with differences in comments and blanks overlooked.

```
COMPARE :
COMPARE : diff        "tod1                     "  "tod2                      "
COMPARE :             line         context         line         context
COMPARE : ---------------------------------------------------------------------
COMPARE :   1 diff at     3          character*8      9          implicit
COMPARE :     same at     3          ^               10          ^
COMPARE :   2 diff at     5          call date (t    12          external
COMPARE :     same at     5          ^               18          ^
COMPARE :   3 diff at     6  write (6,10) tod        22 rite ( 6, 2000 ) tod
COMPARE :     same at     8          ^               24          ^
COMPARE :
COMPARE : Number of differences reported =    3
COMPARE :
```

In this case, `COMPARE` reports that `tod2` has an `implicit` statement and an `external` statement whereas `tod1` does not, and that the target for the write statement differs in the two versions (10 in `tod1`, 2000 in `tod2`). This is perhaps the most useful of all reports, and illustrates the power of being able to ignore selectively various types of differences.

6.2 Comparing declaration contents

The second way *EDITOR* may compare two files is by comparing the contents of their respective declaration lists *without regard to format*. This mode of comparison does not extend into the body of the FORTRAN module containing the FORTRAN executables. The utility of this feature is perhaps best illustrated by an example. In an attempt to align a subroutine with a particular coding style, suppose one wishes to reformat all the declarations at the beginning of a program module from the style illustrated in the file `dec1` to that in file `dec2` below:

Style 1 (file `dec1`):

```
      implicit none
      integer idim, jdim
      parameter (idim=100,jdim=100)
      integer i,j,is,ie,js,je,ieqs(10)
      real s1,v1(idim),w1(jdim),a1(idim,jdim),s2,v2(idim),w2(jdim),
     .a2(idim,jdim)s3,v3(idim),w3(jdim)s4,s5,s6,s7,s8,s9,s10,
     .reqs(10),reqv(3*idim+3*jdim)
c
      equivalence (ieqs(1),i),(ieqs(2),is),(ieqs(3),ie),(ieqs(4),j),
     .(ieqs(5),js),(ieqs(6),je)
      equivalence (reqs(1),s1),(reqs(2),s2),(reqs(3),s3),(reqs(4),s4),
     .(reqs(5),s5),(reqs(6),s6),(reqs(7),s7),(reqs(8),s8),(reqs(9),s9),
     .(reqs(10),s10),(reqv(1),v1(1)),(reqv(idim+1),v2(1)),
     .(reqv(2*idim+1),v3(1)),(reqv(3*idim+1),w1(1)),
     .(reqv(jdim+3*idim+1),w2(1)),(reqv(2*jdim+3*idim+1),w3(1))
      common /comi/ ieqs
      common /comr/ reqs,reqv,a1,a2
```

Style 2 (file dec2):

```
c---------------------------------------------------------------------
c-------------------------- IMPLICIT STATEMENT -------------------------
c---------------------------------------------------------------------
       implicit      none
c---------------------------------------------------------------------
c----------------------------- PARAMETERS -----------------------------
c---------------------------------------------------------------------
       integer       idim    , jdim
       parameter     ( idim =  100, jdim =  100 )
c---------------------------------------------------------------------
c----------------------------- VARIABLES ------------------------------
c---------------------------------------------------------------------
       integer       i       , is      , ie
     1             , j       , js      , je
       real          s1      , s2      , s3      , s4      , s5
     1             , s6      , s7      , s8      , s9      , s10
     2             , s11     , s12
c
       real          v1      (idim), v2      (idim), v3      (idim)
     1               w1      (jdim), w2      (jdim), w3      (jdim)
c
       real          a1      (idim,jdim), a2      (idim,jdim)
c---------------------------------------------------------------------
c----------------------- EQUIVALENCE STATEMENTS -----------------------
c---------------------------------------------------------------------
       integer       ieqs    (  10)
       equivalence
     1   (ieqs( 1),i       ),(ieqs( 2),is      ),(ieqs( 3),ie      )
     2  ,(ieqs( 4),j       ),(ieqs( 5),js      ),(ieqs( 6),je      )
c
       real          reqs    (  20)
       equivalence
     1   (reqs( 1),s1      ),(reqs( 2),s2      ),(reqs( 2),s3      )
     2  ,(reqs( 4),s4      ),(reqs( 5),s5      ),(reqs( 6),s6      )
     3  ,(reqs( 7),s7      ),(reqs( 8),s8      ),(reqs( 9),s9      )
     4  ,(reqs(10),s10     ),(reqs(11),s11     ),(reqs(12),s12     )
c
       real          reqv   (3*idim+3*jdim)
       equivalence   ( reqv    (             1), v1      (1) )
     1             , ( reqv    (        idim+1), v2      (1) )
     2             , ( reqv    (      2*idim+1), v3      (1) )
     3             , ( reqv    (      3*idim+1), w1      (1) )
     4             , ( reqv    (  jdim+3*idim+1), w1      (1) )
     5              , ( reqv    (2*jdim+3*idim+1), w3      (1) )
c-----------------------------------------------------------------------
c---------------------------- COMMON BLOCKS ----------------------------
c-----------------------------------------------------------------------
       common /     comi / ieqs
       common /     comr / reqs    , reqv    , a1        , a2
```

Upon comparing these two declaration modules, *EDITOR* would issue the following difference report:

```
COMPARE :
COMPARE : The following declarations were not found in file "dec1           "
COMPARE :     real          s11
COMPARE :     real          s12
COMPARE :     real          reqs(20)
COMPARE :     equivalence   (reqs(2),s3)
COMPARE :     equivalence   (reqs(11),s11)
COMPARE :     equivalence   (reqs(12),s12)
COMPARE :     equivalence   (reqv(jdim+3*idim+1),w1(1))
COMPARE :
COMPARE : The following declarations were not found in file "dec2           "
COMPARE :     real          reqs(10)
COMPARE :     equivalence   (reqs(3),s3)
COMPARE :     equivalence   (reqv(jdim+3*idim+1),w2(1))
COMPARE :
```

Obviously, several changes other than formatting changes were introduced when `dec1` was recast into `dec2`. Some of the changes may have been deliberate (the addition of variables `s11` and `s12`, increasing the dimension of `reqs` from 10 to 20) while the remainder are probably typos (in the equivalence statements in `dec2`, `reqs(2)` and `w1` appear twice while `reqs(3)` and `w2` do not appear at all).

6.3 The `compare.s` script file

Two source listings may be compared by issuing the following command:

```
csh -v compare.s
```

where `compare.s`, is as follows:

```
#================ SOURCE FILE TO COMPARE TWO LISTINGS =================#
#
#=======================================> Get files from home directory.
if(! -e xedit21) cp  USERID/editor/xedit21 .
#----------------------> Create the input deck for EDITOR, and execute.
rm -f inedit
cat << EOF > inedit
 \$editpar   inname='SOURCECODE1'
          , in2name='SOURCECODE2'
          , ibanner=1, job=5, icompar=1, ndiff=100, ignore=0,0,0     \$
EOF
chmod 755 xedit21
./xedit21
```

A softcopy of `compare.s` may be found in the `editor` directory of `dzeus35.tar` downloaded from `www.ica.smu.ca/zeus3d`.

The first segment retrieves `xedit21` from the user's home directory, if needed. You may wish to add two similar lines to retrieve from their home directories the two files to be compared (`SOURCECODE1` and `SOURCECODE2`). The second segment prepares the input deck for *EDITOR* appropriate for comparing two source listings. In addition to the general namelist parameters described in §2.5, there are four namelist parameters peculiar to `COMPARE`. These are described below.

parameter	description	default
COMPARE (job=5)		
in2name	name of file to be compared with "inname" (char*64).	
icompar	=1 => report first "ndiff" differences =2 => list discrepancies in declarations	1
ndiff	maximum number of differences to report (icompar=1 only).	100
ignore	integer vector indicating types of differences to report (0) or ignore (1). ignore(1): continuation characters; ignore(2): blanks; ignore(3): comments (icompar=1 only).	0,0,0

Some notes:

1. `ignore` is a vector with 3 elements. The first, second, and third elements pertain to continuation characters, blanks, and comments respectively. If any element is 1, that type of difference is ignored, otherwise, it is reported. So, for example, if one wanted to find the differences between two files with differences in continuation characters and comments overlooked (but differences in the number of blanks between words reported), one would set `ignore=1,0,1`.

2. Comparison of declarations may cause an overflow to occur if *EDITOR* was compiled with parameter `k9` set too small. See §9.3 and §A.1.

7 SPLIT: Splitting source code

7.1 Splitting a file

As part of the precompilation process, splitting up a file was discussed in §2.4. One can perform the split outside PRECOM using SPLIT, but it does not include the checking feature that the precompilation split allows. That is to say, no effort is made to check if the file being overwritten *needs* to be updated.

This feature of *EDITOR* is very similar to fsplit. The *EDITOR* SPLIT names each file it creates with the module name and the user-specified extension appended. It will allow the user to specify in which directory all files are to be placed.

7.2 The split.s script file

A source listing may be split by issuing the following command:

```
csh -v split.s
```

where split.s, is as follows:

```
#==== SOURCE FILE TO SPLIT A SOURCE CODE INTO FILES FOR EACH DECK =====#
#
#=======================================> Get files from home directory.
if(! -e xedit21) cp  USERID/editor/xedit21 .
#========================> If necessary, create the directory DIRECTORY.
if(! -e DIRECTORY) mkdir DIRECTORY
#----------------------> Create the input deck for EDITOR, and execute.
rm -f inedit
cat << EOF > inedit
 \$editpar   inname='SOURCECODE'
           , ibanner=1, job=6, idump=1
           , ext='.f', branch='DIRECTORY'                             \$
EOF
chmod 755 xedit21
./xedit21
```

A softcopy of split.s may be found in the editor directory of dzeus35.tar downloaded from www.ica.smu.ca/zeus3d.

The first segment retrieves xedit21 from the user's home directory, if needed. The second segment prepares the input deck for *EDITOR* appropriate for splitting a source listing. In addition to the general namelist parameters described in §2.5, two of the namelist parameters used to control PRECOM are also used by SPLIT. These are described below.

parameter	description	default
SPLIT (job=6)		
branch	character*32 string indicating directory in which separate files are written.	blank
ext	desired extension for files (character*8)	'.src'

8 CONCAT: Concatenating files

EDITOR's `CONCAT` goes beyond the UNICOS utility `cat` by bringing together all files with the same specified extension located in the specified directory *or any of its subdirectories* into a single source file.

8.1 The concat.s script file

Source listings may be concatenated by issuing the following command:

```
csh -v concat.s
```

where `concat.s`, is as follows:

```
#=========== SOURCE FILE TO CONCATENATE FILES INTO ONE FILE ===========#
#
#=======================================> Get files from home directory.
if(! -e xedit21) cp  USERID/editor/xedit21 .
#----------------------> Create the input deck for EDITOR, and execute.
rm -f inedit
cat << EOF > inedit
 \$editpar   outname='OUTFILE'
          , ibanner=1, job=7, idump=1
          , ext='.f', branch='TOP DIRECTORY'                          \$
EOF
chmod 755 xedit21
./xedit21
```

A softcopy of `concat.s` may be found in the `editor` directory of `dzeus35.tar` downloaded from `www.ica.smu.ca/zeus3d`.

The first segment retrieves `xedit21` from the user's home directory, if needed. The second segment prepares the input deck for *EDITOR* appropriate for concatenating source listings. In addition to the general namelist parameters described in §2.5, two of the namelist parameters used to control `PRECOM` are also used by `CONCAT`. These are described below.

parameter	description	default
CONCAT (job=7)		
branch	character*32 string indicating top directory from which files are sought.	blank
ext	common extension of files (character*8).	'.src'

9 Installing *EDITOR*

9.1 Installation

This section describes how to install *EDITOR* on your UNIX-based system. It is assumed that the user has created a fresh directory called, for example, `editor_v2.1_ICA` and, into that directory, downloaded the file `dzeus35.tar` from `www.ica.smu.ca/zeus3d`. Once this tar-file is unpacked, type:

```
ls -FC *
```

and the following should appear on your screen:

```
README       bldlibo*     edit21.tar

editor:
compare.s           edit21.LINUXNAG.f    edit21.s
concat.s            edit21.OS2GNU.f      merge.s
edit21              edit21.OS2WATCOM.f   number.s
edit21.AIX.f        edit21.SUNOS.f       precom.s
edit21.CONVEXOS.f   edit21.SUNOSF95.f    split.s
edit21.LINUX.f      edit21.SUNOSGNU.f    target.s
edit21.LINUXIFC.f   edit21.UNICOS.f

manuals:
edit21_man.ps

nmlst:
dnamelist     dnamelist.s   namelist      namelist.s
```

These are all the files needed to run and operate the program `edit21`, version 2.1 of *EDITOR*.

EDITOR actually manages itself. That is, peppered throughout *EDITOR* are numerous *EDITOR* commands described in §2.1. Other than the `*dk` and `*cd` statements, these are mostly `*if def...*endif` constructs to account for the various operating systems (OS) *EDITOR* supports. Therefore, to get things started, precompiled versions of `edit21` for each OS supported are provided in the directory `editor`, each stripped of all *EDITOR* commands and thus ready for the compiler.

The following instructions are written as though the user were working under SUNOS (SOLARIS). Thus, some translation of the instructions may be necessary for platforms other than SUNOS.

STEP 1: Create a preliminary `edit21` executable, `xedit21`. Since `edit21` manages itself, in principle one needs the `edit21` executable in order to compile it! Thus, a bit of a "boot-strap" process is required to complete the compilation.

Go to directory `editor`.

1.1) If you are operating under one of the supported platforms (*e.g.*, `AIX`, `CONVEXOS`, *etc.*), type:

```
f77 -o xedit21 edit21.<OS>.f
```

where `<OS>` is the appropriate OS tag, and replace `f77 -o` as appropriate. This will create the preliminary `xedit21` executable. Move on to step 2.

1.2 If you are under an OS not represented in the eleven `edit21.OS.f` files in directory `editor`, you will have to modify `edit21.SUNOS.f` as follows. Type:

```
f77 -o xedit21 edit21.SUNOS.f
```

replacing `f77 -o` as appropriate to launch the platform's FORTRAN compiler. This will almost certainly fail, citing calls to unknown subroutines such as `etime`, `time`, `fdate`, and possibly `system`. Under SUNOS, these do the following:

`etime`	returns the cpu time since the beginning of the run in seconds;
`time`	returns the total elapsed wall-clock time in seconds (integer) since the last call to `time`;
`fdate`	returns the date in a 26 character string with the format `Mon Aug 21 14:58:24 2000` (yes, there are only 24 characters, the last two are blank for some reason);
`system`	allows you to make a systems call (such as removing a file from disc) from within the FORTRAN source code at run time.

You will have to find the equivalents to these under your OS, and then adjust the FORTRAN logic in `edit21.SUNOS.f` that depends upon these calls. For example, you may find your system gives a date in a different format, in which case you will have to change the FORTRAN which uses these data. The best way to see where the system dependent stuff is is to look in `edit21` (not `edit21.SUNOS.f`), and scan for the phrase:

```
*if def,SUNOS
```

and then make the necessary changes to `edit21.SUNOS.f` (not `edit21`). Once all the necessary changes have been made and presuming your file is now called `edit21.NEWOS.f` where `NEWOS` is the tag chosen for your OS (typically, the name of your OS in "allcaps"), type:

```
f77 -o xedit21 edit21.NEWOS.f
```

This should produce a preliminary `edit21` executable (`xedit21`) needed through step 4.

Step 2: From directory `editor_v2.1_ICA`, open the script file `bldlibo` in your text editor. The entire file should be platform independent with the possible exception of lines 17 and 18 which invoke the f77 compiler. The options listed are valid for the f77 compilers on both SUNOS and AIX, but may be different for other OSs. The intent of the options on line 17 is to create object files without linking (`-c`), and to invoke the fastest safe optimisation level (`-O4`). The options in the commented out line 18 are for full debugging. The comment before line 17 indicates a bug in the SUNOS compiler at the time of this writing that may not apply elsewhere. Finally, f90/f95 users should make the appropriate changes to lines 17 and 18.

Step 3: From the directory `nmlst`, issue the following commands:

```
csh -v namelist.s
csh -v dnamelist.s
```

These commands will create single- and double-precision versions of the platform-independent namelist libraries (`namelist.a` and `dnamelist.a` respectively). The former is required for `edit21` while the latter is included to link to double precision software that the user may wish to manage with *EDITOR*.

Step 4: If your OS is one of the eleven supported OSs, go to step 5. If not, you must put all the changes you made to create `edit21.NEWOS.f` into the "master copy" of *EDITOR*, namely `edit21`.

From the directory `editor`, open `edit21.NEWOS.f` in one text editing window, `edit21` in another. Without deleting any lines in `edit21`, add your system-dependent changes to `edit21`. Make sure each addition begins and ends with:

```
*if def NEWOS
*endif NEWOS
```

just like the `SUNOS`, `AIX`, *etc.* equivalents do. Follow the `SUNOS` examples already in `edit21` carefully. You should also make certain that the OS tag you invented in step 1 (*e.g.*, `NEWOS`) is unique, and not already used in `edit21` for something else.

Step 5: From the directory `editor`, open `edit21.s` (described in §9.2) and replace the one occurrence of `SUNOS` with your own operating system tag. Save the change. Then, even if you left the tag as `SUNOS`, type:

```
csh -v edit21.s
```

which creates your final working version of `xedit21` (replacing the temporary executable created in step 1), and leaves it in the directory `editor`. You are now ready to go.

9.2 The script file `edit21.s`

The script file `edit21.s` used in the previous section to install *EDITOR* serves as a good example of how the author uses *EDITOR* in practise. For the purposes of this discussion, the file is reproduced below, with an electronic copy of a similar file available in the directory `editor` of `dzeus35.tar` downloaded at `www.ica.smu.ca/zeus3d`.

```
#============= SOURCE FILE TO CREATE THE EDITOR EXECUTABLE =============#
#
#=======================================> Get files from home directory.
if(! -e edit21) cp  USERID/editor/edit21 .
if(! -e xedit21) cp  USERID/editor/xedit21 .
if(! -e namelist.a) cp  USERID/namelist.lib/namelist.a .
#========================> If necessary, create the directory "editor2.1".
if(! -e editor2.1) mkdir editor2.1
#-----------------------------------------------> Create the change deck.
rm -f chgedit
cat << EOF > chgedit
*define SUNOS
*delete par.10,11
      parameter     ( k1=1000, k2=100000, k3=2000, k4=2000, k5=2000
     1               , k6=128, k7=1000, k8=1000, k9=4000 )
**read chged21
```

```
EOF
#========================> Create the input deck for EDITOR, and execute.
rm -f inedit
cat << EOF > inedit
 \$editpar   inname='edit21'
          , idump=1 ,job=3, ipre=1, inmlst=1, iutask=0, safety=0.4
          , chgdk='chgedit'
          , iupdate=1, ext='.f', branch='editor2.1'
          , makename='makeedit', xeq='xedit21'
c         , coptions='-g -C -ftrap=common', loptions='-g'
          , coptions='-fast', loptions='-fast'
          , libs='namelist.a'                                        \$
EOF
chmod 755 xedit21
./xedit21
#=========================================> MAKE the EDITOR executable.
make -f makeedit
```

As with previous examples, the first segment of `edit.s` retrieves the necessary files from the user's home directory, including the *EDITOR* binary executable if not already in the current directory.

The second segment creates a directory on disc called `editor2.1` into which all files split from the master source file `edit21` and object files once compiled are placed.

The third segment generates the change deck `chgedit` which gets merged with the master file `edit21`. This change deck first sets the operating system by defining the appropriate *EDITOR* definition. If there were any *EDITOR* aliases to be set, they might be set here too. Next, the change deck sets values for the parameters described in §9.3. Because of the memory management built into *EDITOR*, it is unlikely that these values would ever have to be changed from those given. Finally, the bulk of the changes to the source code are delegated to a file called `chged21` which, in this example, is *not* merged with `edit21` because of the double asterisk interpreted as a comment by *EDITOR*. Should the user wish to change *EDITOR* for any reason, this is an obvious place to link the changes.

If there were numerous *EDITOR* macro settings, one might consider placing them all in a separate file (called, for example, `edit21.mac`) and then insert the statement `*read edit21.mac` where the `*define SUNOS` statement currently is. In this way, the script file will remain concise, and only those changes which need to be the most accessible (setting the parameter values, for example) would remain in the script file itself.

The fourth segment generates the namelist input data file `inedit` which instructs the current version of the *EDITOR* binary executable `xedit21` how to preprocess `edit21`. In this example, the namelist parameters are set so that `chgedit` will first be merged into `edit21` which is then precompiled (since `ipre=1`) for SUNOS with the namelist replacement feature *on*, the conditional splitting feature *on*, and the micro-tasking feature *off*. All files split from the merged master source code will be placed in the directory `editor2.1` and will have the extension `.f` appropriate for FORTRAN source code. Since `iupdate=1` *and* `makename` are set to a non-blank character string in the input deck, a makefile with the name `makeedit` will be generated. The makefile will use the default compiler (`f77 -c` under SUNOS) and loader (`f77` under SUNOS) with high optimisation (`coptions` and `loptions` both set to `-fast`, appropriate for SUNOS). An optional set of compiler and loader options suitable for debugging is commented out. The library `namelist.a` will be linked with the

EDITOR object code to generate a binary executable called `xedit21`. Finally, this segment executes the current version of `xedit21` and `edit21` is preprocessed.

The fifth segment fires up the makefile `makeedit` generated in the previous segment. This compiles and links the new *EDITOR* code generating a new version of `xedit21` and overwriting the version of `xedit21` which did the preprocessing.

9.3 *EDITOR* parameters

As FORTRAN uses static memory, there are numerous parameters that *EDITOR* requires to set internal array sizes. For the most part, the default settings should be fine for any application on any platform. However, in case the user needs to change any of them, an exhaustive list of `EDITOR`'s parameters, what they limit, and their default values are given below.

parameter	interpretation	default
k1	maximum number of errors to be reported	1,000
k2	maximum number of lines in source code to be processed	100,000
k3	maximum number of origins per module (TARGET)	2,000
k4	maximum number of decks in source code	2,000
k5	maximum number of targets per module (TARGET)	2,000
k6	maximum number of characters per line of source code	128
k7	maximum number of EDITOR definitions	1,000
k8	maximum number of EDITOR aliases	1,000
k9	maximum number of variables within a module	4,000

The purpose of the last parameter `k9` depends on the *EDITOR* function. When comparing declaration lists of two files (§6.2), `COMPARE` will only allow as many `k9` variables of *each* type (real, integer, *etc.*). The namelist replacement feature (§2.2) will allow as many as `k9` variables to be defined in each namelist. Finally, the micro-tasking feature (§2.3) will scope as many as `k9` variables in each nested loop structure it encounters. The default values listed above should be more than adequate for most purposes.

The parameter `k2` sets the maximum number of lines read *at a time* from the user's source file being processed. A source file of *any* arbitrary size may be read. However, if the source code is too long, the file will be read in pieces rather than all at once in order to avoid surpassing the memory available on your machine. Thus, *EDITOR* is built with an effort made toward memory management, and this should be entirely transparent to the user. A 100,000 line source code may be read in and processed all at once (with `k2` set as high as need be) or in various pieces. The end result will be identical. There are a few factors to be considered in choosing the appropriate value of `k2`.

1. *EDITOR* is *slightly* faster if `k2` is large enough to read the entire input file at once, but only slightly.

2. The amount of memory required by *EDITOR* is largely dictated by `k2`. There are four `character*128` arrays dimensioned with `k2`. Thus, the memory required by these arrays for `k2=100000` (the default) is 51.2 Mbytes. It turns out that the size of the *EDITOR* executable with `k2=100000` is 52.4 Mbytes, so it is clear that the `k2` arrays

are the dominant sink of memory. Use these numbers to guide your selection for `k2` and choose as large a value as may be conveniently handled by your machine.

3. The number of lines read by *EDITOR* at a time will be at most `safety*k2`. The variable `safety` may be set in the namelist input data file `inedit` (§2.5 and §9.2) and reflects the fact that the source code is apt to expand during preprocessing. For `NUMBER`, the expansion is minimal (table of contents only), and so `safety=0.9` is probably OK. For `TARGET`, the same. However, for `MERGE` the amount of expansion depends on how much extra coding is being merged with the original file. For `PRECOM` the `*call` statements can result in *substantial* expansion of the source file. Thus, setting `safety=0.4` or smaller may be appropriate. If `safety` is set too high and an overflow results, execution will be aborted and the user will be asked to use a smaller value for `safety` (see §A.1). Since *EDITOR* reads only complete decks and in general, a deck won't happen to end after exactly `safety*k2` lines, *EDITOR* will usually read less than `safety*k2` lines at a time. If there are individual decks with more than `safety*k2` lines, this will create an overflow condition, and execution will abort (§A.1). The user must then do one or more of the following: resubmit the *EDITOR* job with a larger value of `safety` (at the risk of generating overflows when the new files expand); rewrite the source code with smaller decks; or recompile *EDITOR* with a larger value of `k2`. If none of these can be done, *EDITOR* may not be used for the desired task on the chosen machine.

A Error messages

Error messages come in four flavours. The most serious cause *EDITOR* to abort execution and usually require it to be recompiled with higher values for one or more parameters. The second type consist of error messages caused by incorrect *EDITOR* syntax in the files being processed. These are not fatal to *EDITOR* itself, but will mean that the files processed by *EDITOR* will be unusable for their intended purpose. *EDITOR* will insert these error messages directly below the offending line in the output file. Note that original input files supplied by the user are *never* altered by *EDITOR*. The third type are warning messages which are echoed on the terminal screen. In this release, there is only one warning, and it is completely innocuous. Finally, if the user's program is preprocessed with the namelist replacement feature (§2.2), syntax errors in the user's namelist input data file will generate fatal error messages at run time.

A.1 Fatal errors

Fatal error messages which abort *EDITOR* arise under two conditions. Either an overflow has occurred (in which case *EDITOR* may have to be recompiled with one of its parameters set to a higher value), or the file that *EDITOR* was trying to read was not found or was found to be corrupted. All fatal error messages indicate which *EDITOR* subroutine found the problem (of no real use except possibly to *EDITOR* programmers) and what the problem is, followed by the unwelcome message: `ABORTS!`. Internally, the `ABORTS!` message is always followed by a `stop` statement, so this message really means what it says.

These messages are listed here alphabetically, with some descriptive text interleaved where necessary. The first set of messages come from `COMPARE` (§6) and, in particular, when the declarations of two files are being compared. These all require that *EDITOR* be recompiled with a larger value of `k9` (§9.3). Note that n indicates the current value of `k9` as already compiled.

```
COMPARE : Number of character variables exceeded n.  Increase parameter k9.
COMPARE : ABORTS!

COMPARE : Number of data assignments exceeded n.  Increase parameter k9.
COMPARE : ABORTS!

COMPARE : Number of equivalences exceeded n.  Increase parameter k9.
COMPARE : ABORTS!

COMPARE : Number of externals declared exceeded n.  Increase parameter k9.
COMPARE : ABORTS!

COMPARE : Number of integer variables exceeded n.  Increase parameter k9.
COMPARE : ABORTS!

COMPARE : Number of logical variables exceeded n.  Increase parameter k9.
COMPARE : ABORTS!

COMPARE : Number of parameters exceeded n.  Increase parameter k9.
COMPARE : ABORTS!
COMPARE : Number of real variables exceeded n.  Increase parameter k9.
COMPARE : ABORTS!

COMPARE : Number of variables in common exceeded n.  Increase parameter k9.
COMPARE : ABORTS!
```

CONCAT (§8) has one trap which will abort execution should it have trouble reading a particular line (i) in a named file (*filename*). This indicates that the file may be corrupted or is shorter than *EDITOR* anticipated. Check the contents of the file, particularly around the indicated line.

```
CONCAT  : Problem reading line i in file "filename"
CONCAT  : ABORTS!
```

If a user-specified input file isn't actually there, the following message will be issued. Check that *filename* was spelled correctly, or that the file is in the proper directory.

```
OFILE   : Problem opening file "filename"
OFILE   : ABORTS!
```

While *EDITOR* has been designed with considerable effort toward memory management (§9.3), it is still possible for overflows to occur. There are two types of overflow. One can be corrected only by recompiling *EDITOR* with a larger value for the parameter `k2` (§9.3). The other may be corrected by resetting the parameter `safety` in the namelist input data file `indata` (§2.5 and §9.2) to a lower value, then executing *EDITOR* again. Note that the latter case does *not* require recompiling *EDITOR*.

The overflowed text is not written to the intended file, but to an emergency file called "OVERFLOW.TXT" that the subroutine OVERFLOW opens, writes to, and closes before it aborts execution. It may help to examine the contents of this file, but in all likelihood, all one can do is to reset `k2` and/or `safety`. Following are the OVERFLOW fatal error messages. They indicate which subroutine originally detected the overflow and whether to recompile with a higher value for `k2` or to resubmit with a lower value of `safety`. The last message is probably trouble since it implies that enough error messages were inserted to exceed array bounds.

```
OVERFLOW: File overflow in CDECKS.  Increase parameter "k2"
OVERFLOW: Overflowed text file written to file "OVERFLOW.TXT"
OVERFLOW: ABORTS!

OVERFLOW: File overflow in MERGE.  Increase parameter "k2"
OVERFLOW: Overflowed text file written to file "OVERFLOW.TXT"
OVERFLOW: ABORTS!

OVERFLOW: File overflow in MERGE.  Specify lower value for "safety" than x
OVERFLOW: Overflowed text file written to file "OVERFLOW.TXT"
OVERFLOW: ABORTS!

OVERFLOW: File overflow in NMLST.  Specify lower value for "safety" than x
OVERFLOW: Overflowed text file written to file "OVERFLOW.TXT"
OVERFLOW: ABORTS!

OVERFLOW: File overflow in PARALLEL.  Increase parameter "k2"
OVERFLOW: Overflowed text file written to file "OVERFLOW.TXT"
```

```
OVERFLOW: ABORTS!

OVERFLOW: File overflow in PRECOM.  Specify lower value for "safety" than x
OVERFLOW: Overflowed text file written to file "OVERFLOW.TXT"
OVERFLOW: ABORTS!

OVERFLOW: File overflow in TARGET.  Specify lower value for "safety" than x
OVERFLOW: Overflowed text file written to file "OVERFLOW.TXT"
OVERFLOW: ABORTS!

OVERFLOW: In ERRMSG, attempt to insert error message causes a file overflow
OVERFLOW: Overflowed text file written to file "OVERFLOW.TXT"
OVERFLOW: ABORTS!
```

Finally, the subroutine which reads text once a text file is opened (`RFILE`) can fall into traps. *EDITOR* will always attempt to read entire decks or modules of a source code (§9.3). If it cannot (that is, if `safety*k2` is *less* than the number of lines in the largest deck), then execution will abort. In addition, if the file being read is corrupted or smaller than anticipated, a fatal error message will be generated.

```
RFILE   : A module in file "filename" is longer than i lines
RFILE   : Increase parameter "k2"
RFILE   : ABORTS!

RFILE   : Array bounds exceeded while reading file "filename"
RFILE   : Increase parameter "k2"
RFILE   : ABORTS!

RFILE   : Problem reading line i in file "filename"
RFILE   : ABORTS!
```

A.2 Non-fatal errors

There are a variety of error messages that *EDITOR* inserts into the output files should it uncover any *EDITOR* syntax errors or some selected FORTRAN syntax errors. These errors do not abort execution. *EDITOR* simply notes the error, leaves the errant line in the output file (as opposed to syntactically correct *EDITOR* statements which once carried out are expunged from the output file), and moves on. Every effort has been made to ensure that the error message will appear immediately after the offending line in the output file. This requires a rather elaborate accounting scheme which keeps track of every line added or deleted from the file so that the pointers which indicate where the error messages should be inserted are kept up to date. This logic is prone to flaws and, while the author hasn't noted a single case of a misplaced error message in several years of constant use, it is still conceivable that the occasional error message will appear out of context.

Non-fatal error messages may be generated only by `PRECOM` (§2), `MERGE` (§4), or `TARGET` (§5). The appropriate section should be consulted for the correct syntax of the desired operation. Note that the input files are *never* altered by *EDITOR*. Thus, error messages appear in the `.f`, `.m`, or `.t` output files only, not the input files. Nevertheless, corrections should be made to the original input files—making them to the output files where the error messages appear will have no effect.

`CDECKS : **** ERROR 1 **** > 10 nested *calls.  Does a *cdeck call itself?`

This indicates that the user has exceeded the internal *EDITOR* limit of 10 nested `*calls` to common decks. Since it is unlikely such a complex structure would be deliberate, the message suggests that perhaps the calling tree is closed. *e.g.*, perhaps, common deck 1 calls common deck 2 which calls common deck 3 which calls common deck 1, or something to that effect. If so, this should be corrected.

`CDECKS : **** ERROR 2 **** Call made to an unknown common deck.`

A call has been made to an undefined common deck. Misspelling a common deck name in a `*call` statement is the most probable cause for this error.

`COLLECT : **** ERROR 3 **** Expecting a character expression.`

If a `*call` statement appears without a common deck named, or if any other *EDITOR* command is incomplete, this error message will result. Check syntax.

`COLLECT : **** ERROR 4 **** Too many switches defined.`

This indicates that the user has specified more switches (definitions) than allowed by the array bounds. One either needs to define fewer macros, or to recompile *EDITOR* with a larger value of `k7`.

`COLLECT : **** ERROR 5 **** Too many aliases defined.`

This indicates that the user has specified more aliases than allowed by the array bounds. One either needs to set fewer aliases, or to recompile *EDITOR* with a larger value of `k8`.

`COLLECT : **** ERROR 6 **** Too many common decks defined.`

This indicates that the user has specified more common decks than allowed by the array bounds. One either needs to define fewer common decks, or to recompile *EDITOR* with a larger value of `k4`.

`COPY : **** ERROR 7 **** Syntax!  First character is illegal.`

This message is a combined *EDITOR*-FORTRAN syntax error message, and indicates that the only allowed characters in the first column of legal FORTRAN/*EDITOR* statements is a blank, a digit (`0` through `9`), `c`, `C`, or `*`.

`COPY : **** ERROR 8 **** Syntax!  EDITOR sentinel (*) not in column 1.`

This message indicates that the *EDITOR* command does not begin in the first column. *EDITOR* will not attempt to read statements that do not adhere to the precise syntax.

`COPY : **** ERROR 9 **** Syntax!  Extra space after EDITOR sentinel (*).`

This message indicates that the *EDITOR* sentinel (`*`) is not followed immediately by the rest of the command. Again, strict compliance with the syntax is required.

```
CTOI : **** ERROR 10 **** A non-numeric character detected.
```

Some *EDITOR* commands have fields inside which integers are expected. Thus, if a user inadvertently types `*delete sub1.io` where the `io` should have been a `10`, ERROR 10 will be generated.

There is no error message 11.

```
ERRSET : **** ERROR 12 **** Number of issued errors exceeds "k1".
```

Trouble. If one really wants to see all the errors at the same time, go ahead and recompile *EDITOR* with a higher value of `k1`. However, there is probably something really wrong with the input files, and this will have to be corrected. Once the errors are fixed, this error message should go away as well.

```
INOREX : **** ERROR 13 **** Unrecognised EDITOR command.
```

EDITOR didn't recognise what followed the `*` sentinel as a valid *EDITOR* command. Check syntax.

```
INOREX : **** ERROR 14 **** More than ten nested *if statements.
```

EDITOR only allows as many as 10 nested `*if` statements, whether they be `*if define` or `*if alias`. One needs to simplify the nested structure.

```
INOREX : **** ERROR 15 **** Expecting a character expression.
```

See ERROR 3.

```
INOREX : **** ERROR 16 **** Expecting a Boolean .and. or .or..
```

If an *EDITOR* `*if define` statement lists more than one macro, these macros must be separated by an `.and.` or an `.or.`.

```
INOREX : **** ERROR 17 **** Expecting a Boolean .eq. or .ne..
```

An `*if alias` statement must have either an `.eq.` or a `.ne.` in the fourth field.

```
INOREX : **** ERROR 18 **** Dangling *endif statement.
```

More `*endif` statements than the number of pending `*if` statements have been discovered. All `*endifs` must have a corresponding `*if` statement appearing before it in the deck.

```
INOREX : **** ERROR 19 **** Too few *endif statements in previous deck.
```

At least one `*if` statement was not closed by an `*endif` statement by the time the previous deck was closed. All `*if` statements must be closed by an `*endif` statement before the next deck begins. This message is placed immediately after the `*deck` statement of the deck immediately following the deck with the unbalanced `*if` statement(s).

`LIST : **** ERROR 20 **** Unbalanced parentheses.`

A FORTRAN statement with an argument list (`if`, computed `goto`, *etc.*) has unbalanced parentheses. This FORTRAN syntax error is only reported when it interferes with the desired preprocessing (*e.g.*, looking for targets at the end of an `if` statement).

`MERGE : **** ERROR 21 **** Expecting a character expression.`

See ERROR 3

`MERGE : **** ERROR 22 **** Reference made to an undefined deck.`

A `MERGE` edit (`*insert`, `*delete`, or `*replace`) has been issued which refers to a deck not found in the master source code. The most likely reason is that the deckname has been misspelled.

`MERGE : **** ERROR 23 **** Expecting a line number to follow deck name.`

`MERGE` edits require at least one reference to a line number. If no line numbers were given, this error message is issued.

`MERGE : **** ERROR 24 **** Last line number must be greater than first.`

For the `MERGE` edit: `*delete` *filename.n,m*, m must be greater than n.

`MERGE : **** ERROR 25 **** Specified range not found in deck.`

The specified lines in the `*delete` statement were not found in the specified deck. Check the fifth column of the source code formatted by `NUMBER` and make sure the lines do indeed exist.

`MERGE : **** ERROR 26 **** Cannot find specified line.  Was it deleted?`

Probable cause of this error message is a previous `*delete` statement removed the line(s) that the current `MERGE` edit is trying to affect.

`NMLST : **** ERROR 27 **** Missing opening delimiter /.`

The NMLST errors should not be confused with the namelist error messages that may appear during run time of the user's program should syntax errors be found in the namelist input data file. The NMLST errors are those generated by improper syntax of the namelist statement in the user's source code itself. This error message indicates that the opening slash is absent from the namelist designation. See §2.2 for an example of proper namelist syntax.

`NMLST : **** ERROR 28 **** Missing closing delimiter /.`

This error message indicates that the closing slash is absent from the namelist designation. See §2.2 for an example of proper namelist syntax.

`NMLST : **** ERROR 29 **** Too many variables in namelist.  Increase "k9".`

This error message indicates that the user has defined more variables in the namelist statement than may be accommodated by the current setting of the internal parameter `k9`. One may break up the namelist into smaller namelists, or recompile *EDITOR* with a larger value of `k9` (§9.3).

`NMLST : **** ERROR 30 **** Too many vars. in namelist.  Increase "inmlst".`

This error message indicates that the user has defined more variables in the namelist statement than the user-selected value of `inmlst` (§2.5) would allow. One may break up the namelist into smaller namelists, or reset `inmlst` to a higher value.

`NMLST : **** ERROR 31 **** No variables found in namelist.`

There must be at least one valid variable listed in each namelist defined.

`NMLST : **** ERROR 32 **** Unrecognised syntax in dimension statement.`

In replacing the namelist statement with calls to subroutines in `namelist.a`, *EDITOR* must make decisions as to which subroutines to call. In so doing, *EDITOR* scans the declarations at the head of the program module to learn about the attributes (real, integer, *etc.*) and dimensions, if any, of each variable in the namelist. Should it find any FORTRAN syntax error in the declaration list that impedes its scan, *EDITOR* generates this message.

`NMLST : **** ERROR 33 **** Blank "nlsdac" suffix!  Check declarations.`

This error stems (usually) from some syntax error (such as a missing comma) in the declaration list that wasn't detected while the declaration list was being scanned and left the variable attribute undetermined. In this event, *EDITOR* is unable to determine which of the `namelist.lib` subroutines (named `nlsdac`*nn*, where *nn* ranges from `01` to `24`) to call. Check which variable is involved with the statement to which this error message refers and examine how that variable is declared in that program module.

`PRECOM : **** ERROR 34 **** Call made to an unknown common deck.`

See ERROR 2.

`TARGET : **** ERROR 35 **** Unrecognised character in first five columns.`

`TARGET` errors refer always to incorrect FORTRAN syntax which impedes the tidy-up process. This particular error is generated if there is anything other than a digit (0 to 9) in the first 5 columns of any statement that is not a comment, blank, or *EDITOR* statement.

`TARGET : **** ERROR 36 **** Too many targets - Going on to next deck.`

Too many targeted statements in the program module. The tidy-up process cannot continue with current module, so execution continues with the next module. Either break the module up into smaller modules, or recompile *EDITOR* with a larger value of `k5` (§9.3).

```
TARGET : **** ERROR 37 **** Unrecognised syntax.
```

The statement does not conform to ANSI-standard FORTRAN77.

```
TARGET : **** ERROR 38 **** Expecting a numerical target.
```

A FORTRAN statement which is supposed to include a numerical target (*e.g.*, `goto`) did not have a numerical target where one was expected.

```
TARGET : **** ERROR 39 **** Too many origins - Going on to next deck.
```

Too many origins in the program module. The tidy-up process cannot continue with current module, so execution continues with the next module. Either break the module up into smaller modules, or recompile *EDITOR* with a larger value of `k3` (§9.3).

```
TARGET : **** ERROR 40 **** Target defined more than once.
```

Two or more target statements use the same target number.

```
TARGET : **** ERROR 41 **** Origin has no target.
```

The origin refers to a non-existent targeted statement.

```
TARGET : **** ERROR 42 **** Ambiguous targets.
```

This error stems from the `TARGET` feature which replaces `do-enddo` structures with targeted do-loops. This is a badly designed feature because it requires the `enddo` statement to include the do-loop index in order to perform the replacement. Thus the statement `do i=i1,i2` must end on the statement `enddo i`, not just `enddo`. The trouble is that compilers which offer the `do-enddo` extension treat the do-loop index on the `enddo` statement as *optional*, not *mandatory*. Thus the `TARGET` feature to replace `do-enddo`s with targeted do-loops is not as general as it could be. In particular, `enddo` statements which do not echo the do-loop index will result in this error message being issued.

```
TARGET : **** ERROR 43 **** Too many "do"s and/or "goto"s in this deck.
```

The sum of the number of do-loops and the number of `goto` statements (including `goto`s implied by the `err` and `end` parameters in `open` and `read` statements) exceed `ienddo-ibegdo+1`, where `ibegdo` and `ienddo` are namelist input parameters (see §5.1). One must either decrease the number of targets in this module, or increase the difference between `ibegdo` and `ienddo`.

```
TARGET : **** ERROR 44 **** Too many "read"s in this deck.
```

The number of formatted `read` statements exceeds `iendre-ibegre+1`, where `ibegre` and `iendre` are namelist input parameters (see §5.1). One must either decrease the number of read targets in this module, or increase the difference between `ibegre` and `iendre`.

`TARGET : **** ERROR 45 **** Too many "write"s in this deck.`

The number of formatted `write` statements exceeds `iendwr-ibegwr+1`, where `ibegwr` and `iendwr` are namelist input parameters (see §5.1). One must either decrease the number of write targets in this module, or increase the difference between `ibegwr` and `iendwr`.

`TARGET : **** ERROR 46 **** More than` *i* `decks defined.  Part of file lost.`

Program has more modules (subroutines or decks) than can be accommodated by *EDITOR* as compiled. Either reduce the number of modules in the program, or recompile *EDITOR* with a larger value for `k4`.

`PARALLEL: **** ERROR 47 **** Too many variables in do-loop.  Increase "k9".`

The number of variables to be scoped in the nested do-loop structure being micro-tasked is more than can be accommodated by *EDITOR* as compiled. Either reduce the number of variables being used in the offending do-loop, or recompile *EDITOR* with a larger value for `k9`.

`MERGE : **** ERROR 48 **** Too many nested *reads.`

Like the `*call` statement, as many as 10 `*read` statements may be nested. Since it is unlikely that more than 10 nested `*reads` would ever be deliberate, this is probably caused by a closed loop. That is, change deck 1 reads change deck 2 which reads change deck 3 which reads change deck 1, or something to that effect.

`ERRMSG : **** ERROR 999 **** Unspecified error.`

No guidance, other than to say this message should *never* come up.

A.3 Warnings

In this release, there is only one warning message, per se, and this is completely harmless. It is echoed to the user's CRT and is not placed into any of the output files.

`TARGET : **** WARNING **** Increment for deck:` ***decknam*e** `reduced from` *i1* `to` *i2*`.`

If the number of targets found in the current module exceeds `(ienddo-ibegdo)/inc`, where `ienddo`, `ibegdo`, and `inc` are namelist input parameters chosen by the user (see §5.1), then the user-supplied value for `inc` is reduced until the number of targets is less than `(ienddo-ibegdo)/inc`. If choosing `inc=1` still doesn't do it, then error message 43 (or 44, or 45) will result. Note that the reduction of the value of `inc` applies to the current module only. Where ever possible, *EDITOR* will abide by the user's choice for `inc`.

A.4 `NAMELIST` errors

Namelist error messages (as opposed to the `NMLST` error messages described in §A.2) arise only at run time of software that was precompiled with the namelist replacement option turned on. These messages appear if any syntax errors are found in the namelist input disc files. An example of an *EDITOR* namelist error message follows.

```
**********************************************************************
      , q1=1.0e-10, infile='indata' inam=1,1,2,2,3
                                    ^
NAMELIST ERROR  4 ---> unexpected character - check syntax
**********************************************************************
```

The error message echos the offending line in the namelist input data file, places a carat (^) immediately under the offending character in that line, then explains what is wrong. In this case, there is a missing comma. The only problem with *EDITOR* namelist error messages is that only one message can be generated at a time. Each error message aborts execution and thus it may take several tries before all the syntax errors are found and corrected.

Below is a list of the possible namelist errors along with brief descriptions.

```
NAMELIST ERROR 1 ---> column 1 reserved for comment sentinel:  c
```

The rules of namelist input decks must be adhered to exactly. If a character other than a blank, `c`, or `C` appears in the first column anywhere in the input data file, this message will be issued.

```
NAMELIST ERROR 2 ---> column 2 reserved for $ delimiter
```

If a character other than a blank or `$` appears in the first column anywhere in the input data file, this message will be issued.

```
NAMELIST ERROR 3 ---> variable not found in namelist
```

A variable is being set which was not part of the namelist declaration in the source code. The usual reason for this error is a misspelled variable.

```
NAMELIST ERROR 4 ---> unexpected character - check syntax
```

The usual reason for this message, as in the example above, is a missing comma. Check the syntax of the offending line carefully.

```
NAMELIST ERROR 5 ---> invalid logical expression
```

A logical variable has been assigned a value other than `.true.`, `.t.`, `.false.`, or `.f.` (including the periods).

```
NAMELIST ERROR 6 ---> namelist does not exist, or is out of sequence
```

The source code attempts to read a namelist not found in the namelist input data file. This is usually caused by a misspelled namelist name or a namelist that appears out of order in the data file. Note that the order of the namelists in the data file must be the same as the order in which the source code reads them.

`NAMELIST ERROR 7 ---> error reading input deck`

Does the namelist input data file exist on disc in the directory in which the executable was executed?

`NAMELIST ERROR 8 ---> variable not declared as a vector`

A scalar is assigned values as though it were a vector.

`NAMELIST ERROR 9 ---> next namelist begun before closing quote found`

Character assignments may run over several lines. Thus, if a namelist opening sentinel `$` is found in the second column of a line before the closing quote of the current character assignment is found, this message results. A common cause of this message is if the closing quote has been inadvertently shoved beyond the 72nd column, or if the closing quote is a single (double) quote while the opening quote is a double (single) quote.

`NAMELIST ERROR 10 ---> missing opening quote`

If the first non-blank character after the `=` in a character assignment is not a single or a double quote, this message is issued.

`NAMELIST ERROR 11 ---> blank data`

A variable is assigned a null field.

`NAMELIST ERROR 12 ---> premature end of file`

File appears to have ended before the closing sentinel `$` of the current namelist was found. Common cause is if the closing sentinel of the last namelist was inadvertently shoved beyond the 72nd column.

`NAMELIST ERROR 13 ---> missing $ sentinel to close namelist`

Next namelist has begun (a `$` sentinel was found in column 2) before the closing sentinel of the current namelist was found. Common cause is if the closing sentinel is inadvertently shoved beyond the 72nd column.

`NAMELIST ERROR 14 ---> exponent must not exceed 999`

The *EDITOR* namelist will not permit exponents in real variable assignments to exceed 999 (or be lower than -999). Obviously, a 32-bit word machine will have even more stringent limits.

`NAMELIST ERROR 15 ---> no more than 15 digits in single precision`

The tone of this message is a fossil of the days when *EDITOR* ran only under UNICOS, and where single precision was 15 significant digits (double precision on most other platforms). The *EDITOR* namelist will only allow as many as 15 significant figures to be specified in a real variable assignment.