

Using GEMPACK Subroutines in your Fortran programs

W. Jill Harrison, J. Mark Horridge and Ken Pearson

Centre of Policy Studies, Monash University, Clayton Vic 3800, Australia

28 April 2005

Abstract

General-purpose software packages such as GEMPACK provide tools which allow modellers to concentrate on the economic issues without needing to write their own software. However, it may occasionally be necessary to write programs to carry out some specialised tasks which cannot be carried out using the general-purpose tools provided. Even then, programmers with access to a Source-code version of GEMPACK can make use of the many Fortran routines supplied with GEMPACK.

This paper gives an introduction to using GEMPACK subroutines in writing your own Fortran programs. It gives a brief overview of some of the ideas behind the GEMPACK code such as error handling and to some of the routines available for opening, reading and writing Header Array files. By using standard GEMPACK routines, you can avoid knowing, in great detail, the structure of GEMPACK file types such as Header Array files.

We provide and describe in detail two typical programs which illustrate the techniques involved. The first program reads data from a non-GEMPACK text file, modifies it and then writes the modified data to a Header Array file. The second program reads some simulation results from Header Array files and calculates and writes arrays showing the differences between these results.

The example programs we provide have been chosen because these are two of the tasks for which it may occasionally be necessary to write programs. However, because we want to emphasise how you can use the existing GEMPACK routines to do much of your work, we have chosen to present somewhat simplified versions of the actual tasks which may confront a modeller. For example, in the first example program,

- a) we present rather simple versions of the complexities which may be necessary in order to read and modify data taken from a non-GEMPACK source.
- b) we read, modify and write only one array (whereas a real-life program would probably have to deal with several arrays).

As well as the two example programs, we provide information as to where you can find the source code for different GEMPACK subroutines, how they are organised and how you can compile and link your programs to the GEMPACK libraries.

We also provide templates MODELPRG for a main program (see section 4) and MODELSUB for a subroutine (see section 5). These contain the standard declarations and calls. When we develop a new main program or subroutine, we always start from these. We recommend that you do the same.

Once you have become familiar with the material here, you will be in a good position to understand the code in any of the GEMPACK utility programs (those described in GEMPACK document GPD-4).

1	INTRODUCTION.....	1
1.1	Fixed or Free Source From.....	1
1.2	Upper or Lower Case.....	2
1.3	Fortran Documentation.....	2
2	FIRST EXAMPLE – WRITING DATA AS HEADER ARRAYS.....	3
3	SECOND EXAMPLE – READING SOL FILES AND WRITING DIFFERENCES.....	5
4	STRUCTURE OF A MAIN PROGRAM.....	7
5	STRUCTURE OF A SUBROUTINE.....	8
6	STANDARD FEATURES USED IN GEMPACK PROGRAMS.....	9
6.1	Error Handling – ERRMES.....	9
6.1.1	Routines Called After a Fatal Error.....	10
6.2	GPTO Routines for Writing to the Terminal.....	11
6.2.1	Subroutine GPTOG.....	11
6.2.2	Subroutines GPTO, GPTO1 and GPTO2.....	12
6.3	GPIN Routines for Reading from the Terminal.....	12
6.4	Various Modules.....	13
6.5	Subroutine GPSTRT.....	13
6.6	Subroutine GPSTOP.....	13
6.7	Splitting Long Records Using Machine-Specific Parameters.....	13
6.7.1	Finding Safe Numbers for One Physical Record.....	14
6.8	Unit Numbers – Routines GPUNI2 and GPUNIT.....	14
7	READING AND WRITING HEADER ARRAY FILES.....	16
7.1	Writing a Real Array.....	16
7.1.1	Writing a Real Array Without Set and Element Labelling.....	16
7.1.2	Writing a Real Array With Set and Element Labelling.....	17
7.2	Reading a Real Array.....	18
7.2.1	Reading, Not Set and Element Labelling, Checking Sizes.....	18
7.2.2	Reading, Including Set and Element Labelling, Checking Sizes.....	19
7.2.3	Examples of Set and Element Labelling Information Returned.....	21
7.2.4	Reading, Not Set and Element Labelling, Returning Sizes.....	21
7.2.5	Reading, Including Set and Element Labelling, Returning Sizes.....	22

7.3	Opening and Closing Header Array Files	23
7.3.1	Opening a Header Array File.....	23
7.3.2	Closing an Old Header Array File.....	23
7.3.3	Closing a New Header Array File.....	24
7.3.4	Writing Creation Information and History	24
8	ALLOCATING MEMORY FOR ARRAYS	25
9	WHERE TO FIND THE GEMPACK SUBROUTINES	25
10	COMPILING AND LINKING YOUR PROGRAMS	25
10.1	Using the LF95 Compiler for Code Development	26
11	REFERENCES.....	26

1 Introduction

General-purpose software packages such as GEMPACK provide tools which allow modellers to concentrate on the economic issues without needing to write their own software. However, it may occasionally be necessary to write programs to carry out some specialised tasks which cannot be carried out using the general-purpose tools provided. Even then, programmers with access to a Source-code version of GEMPACK can make use of the many Fortran routines supplied with GEMPACK.

This paper gives an introduction to using GEMPACK subroutines in writing your own Fortran programs. It gives a brief overview of some of the ideas behind the GEMPACK code such as error handling and to some of the routines available for opening, reading and writing Header Array files. By using standard GEMPACK routines, you can avoid knowing, in great detail, the structure of GEMPACK file types such as Header Array files.

We provide and describe in detail two typical programs which illustrate the techniques involved.

1. The program EXWHA1.FOR reads data from a non-GEMPACK text file, modifies it and then writes the modified data to a Header Array file.
2. The program EXRES1.F90 reads some simulation results from Header Array SOL files (produced by running SLTOHT) and calculates and writes arrays showing the differences between these results.

The example programs we provide have been chosen because these are two of the tasks for which it may occasionally be necessary to write programs. However, because we want to emphasise how you can use the existing GEMPACK routines to do much of your work, we have chosen to present somewhat simplified versions of the actual tasks which may confront a modeller. For example, in the first example program,

- a) we present rather simple versions of the complexities which may be necessary in order to read and modify data taken from a non-GEMPACK source.
- b) we read, modify and write only one array (whereas a real-life program would probably have to deal with several arrays).

As well as the two example programs, we provide information as to where you can find the source code for different GEMPACK subroutines, how they are organised and how you can compile and link your programs to the GEMPACK libraries.

We also provide templates MODELPRG for a main program (see section 4) and MODELSUB for a subroutine (see section 5). These contain the standard declarations and calls. When we develop a new main program or subroutine, we always start from these. We recommend that you do the same.

Once you have become familiar with the material here, you will be in a good position to understand the code in any of the GEMPACK utility programs (those described in GEMPACK document GPD-4).¹

1.1 Fixed or Free Source Form

Fortran 90 programs can be written using either fixed or free source form.

Fixed source form is the older Fortran 77 (or even Fortran 4) form. Columns 7-72 of each line are used for Fortran statements. Columns 1-5 are for statement labels. Column 6 may be used

¹ Though not to understand the code of TABLO, GEMSIM and TABLO-generated programs which are more complicated.

to indicate a continuation line. The GEMPACK routines are written using this style (since most were originally written to conform to the Fortran 77 standard).

In free source form, there are no restrictions on where a statement can appear on a line. A line can be up to 132 characters long. If the last non-comment character in a line is an ampersand &, this indicates that the line is to be continued on the next line.

When you write a Fortran 90 program which uses GEMPACK routines, you can choose to write your program in either form. To illustrate the difference, our first example program EXWHA1.FOR is written in fixed form while our second EXRES1.F90 is written in free form.

1.2 Upper or Lower Case

You can use either or a mixture. But different case combinations are treated as the same identifier (for example, NewSizes, NEWSIZES and newsizes all represent the same identifier).

We find that mainly lower case with capital letters used to indicate beginnings of words (for example, NewSizes or KnowAcDim) easiest to read and the best for self documentation. With Fortran 90 you can use long names (up to 31 characters) for identifiers. Long names provide better documentation than short ones (for example, KnowAcDim is better than KNACDM).

1.3 Fortran Documentation

We recommend that you always conform to the Fortran 90 (or 95, or 2000) standard when you write Fortran programs. There are many good books available as guides to writing Fortran programs. A good starting point for detailed information about Fortran is the Language Reference manual supplied with your Fortran compiler.

2 First Example – Writing Data As Header Arrays

You can find the code for our two example programs on the web at address

<http://www.monash.edu.au/policy/gpusesub.htm>

At this address you will find

- **EXWHA1.ZIP** and **EXRES1.ZIP**. These contain the two examples programs and associated files needed to run them.
- a PDF version of this document **USEGPSUB.PDF**.
- **MODELPS.ZIP** which contains templates MODELPRG and MODELSUB. These templates for a main program and a subroutine are discussed in sections 4 and 5 below respectively.

The code and associated input files for the first example are in file EXWHA1.ZIP. Please extract the files from inside this ZIP file.

Look at the source code of EXWHA1.FOR (for example, in TABmate which provides syntax highlighting for Fortran programs). You will see the following parts. We elaborate on these different parts later in this document.

- Declarations section.
- Call to GEMPACK routine GPSTRT to start the executable code.
- Section where the old and new files are opened.
- Call to user routine GetSizes which reads the original of the array to be read and sets up the size of the new array to be written.
- Allocate arrays now that the sizes are known.
- Call to user routine GetNames which reads the original commodity names.
- Call to user routine GetInputData which reads the original data.
- Call to user routine ModifyData which modifies the data into the form to be written.
- Call to user routine WriteData which writes the modified data to the Header Array file. This also sets up the commodity names for the modified array to be written.
- Section where the old and new files are closed.
- Call to GEMPACK routine GPSTOP to stop the program.
- Section handling errors. This ends the source of the main program.
- Separate code for the various user routines GetSizes etc. For example, the code of the WriteData subroutine calls the GEMPACK routine PUT7RE to write the data to a Header Array file.

In this first look at EXHWA1.FOR you should also notice

- the GEMPACK error handling. After each call to a subroutine, there is the line of code

```
if(errmes(1:1).eq.'E') goto 15000
```

which checks if a fatal error has occurred and jumps to the error handling part if so.
- the use of GPTO routines (for example GPTO1) to write prompts to the terminal. In order to get access to all the GEMPACK features such as the use of STI and LOG options, all output to the terminal must go through the different GPTOx routines.

Next you might like to run this example on the simple input file we have supplied. Please extract files EXWHA1.DAT and EXWHA1.STI from EXWHA1.ZIP. To carry out the steps below, go to a DOS box in the directory in which you have placed these files and EXWHA1.FOR.

1. To make an executable image EXWHA1.EXE from EXWHA1.FOR, issue the command

```
ltgnomod exwha1
```

2. The example data file EXWHA1.DAT has the following lines:

```
4
agriculture
food
mining
manufacture
100.1 120.0 230.1
340.2
```

3. Run EXWHA1.EXE taking inputs from Stored-input file EXWHA1.STI by typing the command

```
exwha1 -sti exwha1.sti
```

This reads the data in EXWHA1.DAT (data for the 4 commodities shown), modifies it by aggregating the values for the last two commodities and then writes the modified data to output Header Array file exwha1.har.

4. Finally, look at output file exwha1.har in ViewHAR to check the results.

Now you might like to look at the different sections below which elaborate on the different parts of the code in EXWHA1.FOR. For example,

- see how the outline in MODELPRG in section 4 is adapted to make the main program in EXWHA1.FOR,
- see how the outline in MODELSUB in section 5 is adapted to make the subroutines GetSizes and GetNames in EXWHA1.FOR,
- check that ERRMES is tested after each subroutine call (see section 6.1),
- note how routine GPTO1 (see section 6.2.2) is used to give the prompt for the file names in the main program part of EXWHA1.FOR,
- how routine PUT7RE is called to write the array as a Header Array in subroutine WriteData attached to EXWHA1.FOR,
- how IOSTAT values is used for error trapping for reads in routines GetSizes, GetNames and GetInputData. Note the jumps to statements numbered 23500 and 23700 in appropriate cases.

3 Second Example – Reading SOL Files and Writing Differences

The code and associated input files for this example are in file EXRES1.ZIP. Please extract the files from inside this ZIP file.

Look at the source for of EXRES1.F90 in TABmate.

- You will see at the start and near the end of the main program various parts similar to those described for the first example (see section 2).
- In this case, there is only one subroutine called ReadWriteOneHeader attached at the end of the source code. For each designated header in the two SOL files, this subroutine
 - ❖ reads the values from the two files,
 - ❖ reads the set and element information from the first input file,
 - ❖ calculates the differences between the two sets of results and
 - ❖ writes these differences out to the same header (with a different long name) on the output Header Array file.
- If you look at the code for subroutine ReadWriteOneHeader you will see
 - ❖ a call to routine HAHMD1 which find how many real numbers are stored at this header,
 - ❖ allocate statements to allocate appropriate amounts of memory to the arrays used to read in the values and set elements and to write out the differences,
 - ❖ a call to routine GRVADE to read the data and also the set and element information from the first input file,
 - ❖ a call to routine GTRVAD to read the data (but not the set and element information) from the second input file. [It is reasonable for this program to assume that the set and element labelling on the second file is the same as on the first file.]
 - ❖ a DO loop where the differences are calculated,
 - ❖ a call to routine PUT7RE to write the differences to the output Header Array file.

Next you might like to run this example on the simple input file we have supplied. Please extract the various files from EXRES1.ZIP. To carry out the steps below, go to a DOS box in the directory in which you have placed these files and EXRES1.F90.

1. To make an executable image EXRES1.EXE from EXRES1.F90, issue the command²

```
ltgfnmod exres1
```

2. The input files are SJLB.SOL and SJLB2I.SOL. These are SOL files produced by running GEMPACK program SLTOHT on the two Solution files SJLB.SL4 (produced using the standard SJLB.CMF) and SJLB2I.SL4 which is produced using SJLB2I.CMF – same as SJLB.CMF except that it uses two subintervals. Of course the numbers at corresponding headers in these two files are not much different.
3. Run EXRES1.EXE taking inputs from Stored-input file EXRES1.STI by typing the command

```
exres1 -sti exres1.sti
```

² The command ltgfnmod is used here since it works for source code which is in free form. Note from section 2 that the similar command ltnomod is used for fixed-form source code.

This program reads the data at headers "0002" and "0007" in SJLB.SOL and SJLB2I.SOL, calculates the differences and writes these differences out to the same headers in the output Header Array file sjlbdiff.har.

4. Finally, look at output file sjlbdiff.har and the input files in ViewHAR to check the results.

Now you might like to look at the different sections below which elaborate on the different parts of the code in EXRES1.F90. For example,

- the statements

```
HeadArray(1)='0002' ! p_PC results from SJLB.SOL  
HeadArray(1)='0007' ! p_XC results from SJLB.SOL
```

which tell the program which headers to operate on.³

- the code in subroutine reads values into vectors InputArray1 and InputArray2. This is a flexible way of writing code which can handle arrays of different dimensions. For example, this code works equally well for the 1-dimensional array of p_PC results and the 2-dimensional array of p_XC results.

³ If you want to carry out some operation on all headers on a Header Array file, you can use subroutine LSTHDS or LSTHD2 to find out information about all these headers. Then you can loop through the headers. GEMPACK main program SEEHAR is an example.

4 Structure of a Main Program

We begin from the template main program **MODELPRG** when starting a new main program. [MODELPRG is supplied in zip file MODELPS.ZIP.] MODELPRG is just a skeleton of a main program. Using this template as a starting point will give your program a regular order and structure.

Open the file `MODELPRG` in TABmate. You will see

- a place to specify a program name after the Fortran Keyword PROGRAM.
- a place to explain the acronym used as the program name.
- a place to write a few lines of comments describing the purpose of the program.
- a place to include history information as the program is developed over several years.
- a place for "USE" statements to indicate which Fortran Modules are used. Each main program needs to use module called GPMAIN1. See section 6.4 for information about this module.
- a place for declarations of variables PRNAME, PROGNV, PRVERI and DATES.
- a place to declare local variables.
- a call to routine GPSTRT which is required at the start of each GEMPACK main program. See section 6.5 for more information.
- a call to routine GPSTOP which is required at the end of each GEMPACK main program. See section 6.6 for more information.
- code to handle fatal errors – see the calls to routines PERET, PERETM, GPPGER and finally GPSTOP. These are vital parts of the error handling mechanisms described in section 6.1.

The real code for each main program goes between the GPSTRT and GPSTOP calls. You can call other GEMPACK subroutines anywhere in your program. However they will not work as intended unless you have used GPMAIN1 in your program and unless you have called the starting routine GPSTRT and the stopping routine GPSTOP.

The two example programs EXWHA1.FOR (section 2) and EXRES1.F90 (section 3).are both based on the template MODELPRG.

5 Structure of a Subroutine

We begin from the template subroutine **MODELSUB** when starting a new subroutine.
[MODELSUB is supplied in zip file MODELPS.ZIP]

Open the file **MODELSUB** in TABmate. You will see

- the place at the start to put the subroutine declaration.
- a place to explain the acronym and a place for the history of the routine.
- a place for "USE" statements to indicate which Fortran Modules are used. For example, many subroutines need to use module called GPBAS1. See section 6.4 for information about this module.
- places to declare the variables used in the subroutine statement. We separate these into
 - ❖ Input (not altered) – use "intent(in)" for these.
 - ❖ Output – use "intent(out)" for these.
 - ❖ Input/Output – use "intent(in out)" for these.
 - ❖ Working Variables and Arrays (not always needed)
- a place to declare the variable SUBNAM (name of this subroutine) – this is needed for the error-handling part of the subroutine near the end.
- a place to indicate which other subroutines are called (EXTERNAL and INTERNAL statements) and a place to refer to other similar routines.
- a place to declare local variables.
- a place where we list important statement labels – we always use the same statement numbers for most of these in different routines.
- a place (essentially empty) for the code to do the actual work of the subroutine (EXECUTABLE CODE).
- a place for normal (that is, error-free) return
- places to trap for various standard sorts of errors.

Have a second look at one or more of the subroutines attached to examples EXWHA1.FOR (section 2) and EXRES1.F90 (section 3). Subroutines in EXWHA1.FOR such as GetSizes, and GetNames are firmly based on the template MODELSUB while the EXRES1 subroutine ReadWriteOneHeader is less formal but still preserves all the important elements and standard features.

Most of the points above are fairly standard for any subroutine. The difference for subroutines calling GEMPACK subroutines is in the error handling using ERRMES and the traps for different sorts of errors at the end of the subroutine. See section 6.1 for details of error handling. There are many different error traps included in MODELSUB. However in writing your subroutine you only need to retain the ones you have used in that subroutine.

6 Standard Features used in GEMPACK Programs

You need to be aware of certain standard features of GEMPACK code, as described below, in order to make effective use of GEMPACK routines in your programs.

6.1 Error Handling – ERRMES

This section describes the Error Handling mechanism in GEMPACK so that, if an error occurs, programs

- exit gracefully, and
- show a traceback as to where the error occurred.

Nearly every subroutine (and function) includes the argument ERRMES as the last argument in its calling sequence. This is a CHARACTER*70 string used for error handling and for handling warnings.⁴

If a fatal error occurs, the subroutine which first recognises this error must set the first character of ERRMES to E (upper case E) and usually puts a context-specific string describing the error into the rest of ERRMES. It must then call subroutine PERETM and then exit.

Whenever a routine is called, the calling routine MUST TEST ERRMES(1:1) immediately after the call. If it finds ERRMES(1:1) is equal to E, it must call routine PERET immediately and then exit.

Example.

In the code below (taken from program MKHAR) you see two examples of the standard use of ERRMES.

1. The value of ERRMES(1:1) is tested after the call to OPFILE (which tries to open a file). This test of ERRMES is to exit gracefully if a fatal error has already been set inside OPFILE.
2. There is the use of ERRMES inside the “IF(.NOT.OPENOK) THEN” block. Here the programmer has decided that a fatal error is appropriate if OPENOK is false after the call to OPFILE (that is, if the file has not been opened). Accordingly, ERRMES is assigned a value (whose first character must be E, and the rest of which describes the error) and the code jumps to statement number 15500.

```
CALL OPFILE( OLDUN, 'old', 'formatted', OLDFN, OPENOK, ERRMES)
IF( ERRMES(1:1).EQ.'E') GOTO 15000
IF(.NOT.OPENOK) THEN
    ERRMES='E-UNABLE TO OPEN OLD FILE'
    GOTO 15500
END IF
```

The example shows two statement numbers which are always used (by convention) when processing ERRMES.

1. Statement number **15500**. This is where to jump to immediately after ERRMES is set to hold a fatal error. At statement number 15500, routine PERETM is called.

⁴ ERRMES has length LLERRMES where LLERRMES is declared in module GPBAS1 (see section 6.4).

2. Statement number **15000**. This is where to jump to immediately if ERRMES has been set fatal during a call to a routine (or function). At statement number 15000, routine PERET is called.

User documentation about program errors and tracebacks can be found in section 5.7 of GPD-1.⁵

6.1.1 Routines Called After a Fatal Error

Before the final exit in the case of a fatal error, various cleaning up routines may be called. For example, these routines may

- clean up currently open files. These files will be closed. If they are new files which are incomplete, they may be deleted.
- set an operating-system flag to indicate an error. This will be done in routine GPSTOP which is the only routine allowed to execute a STOP statement.

Only certain GEMPACK routines should be called after a fatal error is set. These are the only ones which are designed to operate in the presence of ERRMES with first character equal to E. Examples of such routines are

- CLSEQ for closing files.
- GPTOWE which is still allowed to write to the terminal in the presence of a fatal error. The other GPTO routines used for writing to the terminal (for example, GPTO1, GPTOG – see section 6.2) should not be called after a fatal error is encountered.

These routines need careful writing. For example (see CLSEQ) they may use a logical variable ISFATL in place of the normal tests of ERRMES after they call other routines.

You can find all of the routines which can operate in the presence of a fatal error by searching the SOURCE subdirectory (see section 9) for the string “one of the routines that is called”.

⁵ In "official" GEMPACK programs and subroutines, there is also statement number 15300 to trap for internal programming errors. At statement number 15300, the routine GPPGER (which asks users to bundle up the relevant files and send them to us) is called. You should not use 15300 or routine GPPGER in your programs since we do not want to be responsible for your internal program errors.

6.2 GPTO Routines for Writing to the Terminal

GPTO stands for "GemPack Terminal Output" and GPTO subroutines are the basis of the LOG files generated by GEMPACK programs. You will need to use these routines if you want to write to the terminal to assist with debugging new code or problems.

All output to the terminal must be done via calls to one of the routines

GPTO, GPTO1, GPTO2 or GPTOG

This is essential if you want the output to get into the LOG file.

6.2.1 Subroutine GPTOG

GPTOG is the most powerful. You can use it to output to the terminal strings containing at most 3 CHARACTER variables, at most 2 INTEGER variables and at most 2 REAL variables. The calling sequences for GPTOG is

```
call GPTOG('t',0,
*      MES1, MES2, MES3,
*      STR1, STR2, STR3,
*      INT1, INT2, REAL1, REAL2, ERRMES)
```

Here the MES_i (MES1, MES2 or MES3) are strings which can contain special characters including

- **%s** (string) or **%t** (string with trailing blanks omitted). The first %s or %t in the MES_i refers to STR1, the second %s or %t refers to STR2 and the third to STR3.
- **%i** (integer). The first %i in the MES_i refers to INT1 and the second %I to INT2.
- **%r** (real). The first %r in the MES_i refers to REAL1 and the second to REAL2.
- **\n** for a new line
- **\b** for an optional break (lines sent to the terminal are limited to 79 or 80 characters)

If you want one of the MES_i to do nothing, put '**\x**' in its place – see the examples below.

These special characters cannot be used in STR1, STR2 or STR3.

By convention we use

- ' ' for any unused STR_i arguments,
- 0 for any unused INT_i arguments and
- 0.0 for any unused REAL_i arguments. [Do not use just 0 for any unused REAL_i arguments since our debugging LF95 compiler options will object.]

Example

```
call GPTOG('t', 0,
*      '\n Set %t has %i elements.\n\n',
*      '\x', '\x', stnam(kst), ' ', ' ',
*      ssz(kst), 0, 0.0, 0.0, errmes)
```

The first **\n** puts one extra blank line before. The ending **\n\n** puts two blank lines at the end. This call could be made shorter using routines GPTO1 (see the next subsection) since it only uses one CHARACTER variable and one INTEGER variable.

6.2.2 Subroutines GPTO, GPTO1 and GPTO2

GPTO, GPTO1 and GPTO2 are used for simpler calls.

- Use GPTO2 if you want to output at most 2 CHARACTER variables and at most 2 INTEGER variables.
- Use GPTO1 if you want to output at most 1 CHARACTER variable and at most 1 INTEGER variable.
- Use GPTO if your output does not need to access any CHARACTER or INTEGER variables.

Examples.

```
call GPTO('Calling TGCNST', errmes)
call GPTO1('Calling %t', subnam, 0, errmes)
call GPTO1(
*   '\n Set %t has %i elements.\n\n',
*   stnam(kst), ssz(kst), errmes)
call GPTO2(
*   '\n The set "%t" is set number %i. It has size %i.',
*   'x', stnam(kst), ' ', kst, ssz(kst), errmes)
```

If you need to output one or more REAL variables, you must use GPTOG.

6.3 GPIN Routines for Reading from the Terminal

All input from the terminal must go through subroutine **GPINC2**. This is the mechanism by which Stored-input files work.

The routine GPINC2 reads a single character string (line of text) from the terminal.

It is not essential to call GPINC2 directly. There are several higher level routines (which call GPINC2 indirectly) which can be called to process input from the terminal. These routines include

- GPINNT and GPINRL which read a single integer or real number respectively from the terminal,
- GPINI2 and GPINIA which read an integer array from the terminal,
- GPINRA which reads a real array from the terminal.

The calling sequences for these routines are documented in their source code – see section 9.

6.4 *Various Modules*

GEMPACK uses a number of standard Fortran modules for declarations of variables and parameters (such as length of file names, number of characters in a Coefficient name) which are used in many different routines. You can see the source of these modules in the MODULES subdirectory of the directory in which your Source-code GEMPACK is installed. You might like to look at the code for the various modules mentioned below as you read about them.

Important modules include

GPBAS1. This contains declarations of Fortran parameters (constants) such as MMLFN (maximum length of a file name), LLHD and LLNAME (length of a header and long name on a Header Array file). It also includes the declaration of LLERRMES which is the length of the character variable ERRMES used for error handling (see section 6.1). Many subroutines USE this module.

GPMAIN1. This contains declarations used in all GEMPACK main programs. The arrays GPRCON and GPRCNI are used to control many of the features common to GEMPACK programs (such as whether or not the GEMPACK program is writing a LOG file, whether or not it is echoing the names of files being opened).

LENGTHS. For example, this contains Fortran parameters (constants) giving the lengths of Coefficient names (LLCS), set names (LLST), set elements (LLSTEL) and Variables (LLVC).

6.5 *Subroutine GPSTRT*

The important subroutine GPSTRT must be called at the start of each GEMPACK main program. It initialises many different parts of the GEMPACK system. It also processes the command line (handling things such as "-cmf" and "-sti"), processes the basic program options (is USEBOP is true in the call). You won't find the code for this routine in the SOURCE subdirectory.

6.6 *Subroutine GPSTOP*

The subroutine GPSTOP must be called just before you want to stop. It is only called in the main program, not in subroutines, so that a trace-back is given in the case of a fatal error. This routine closes the LOG file (if you are producing one). It also sets the DOS ERRORLEVEL to 1 if a fatal error occurs.

6.7 *Splitting Long Records Using Machine-Specific Parameters*

This is about GEMPACK routines which write an array to a binary file (such as a Header Array file).

GEMPACK routines were written to cater for the case where the records produced by the Fortran write statements above may, in some cases, be longer than the length of records supported by a particular compiler. The GEMPACK routines split the above logical records into one or more physical records. Within GEMPACK, information is available from the routine GPBNF1 (see section 6.7.1) about safe numbers of integers, reals and characters allowed on one physical record.

Splitting into one or more physical records is important. For example, the Absoft Fortran compiler supported by GEMPACK in the 1990s only allowed records of at most 1024

characters. So, for example, it was vital to split arrays of more than 256 reals into several physical records.

If you write an array of reals, integers or character strings to a binary file, you must use one of the low level routines described here or a higher level writing routine such as a Header Array write (which calls one of these routines). You must not use a direct Fortran write statement.

6.7.1 Finding Safe Numbers for One Physical Record

Subroutine GPBNF1 can be called to find out safe numbers of reals, integers and characters to write to one physical record. This number may depend on the compiler and/or operating system which is why routine GPBNF1 is in the NPORT group of subroutines.

The calls are

```
call GPBNF1('mmrlr', mmr, errmes)
call GPBNF1('mmrli', mmi, errmes)
call GPBNF1('mmrlch', mmc, errmes)
```

These subroutines return

MMR – safe number of reals to put on one physical record.

MMI – safe number of integers to put on one physical record.

MMC – safe number of characters to put on one physical record.

These values MMR, MMI and MMC can then be passed to the relevant routine which does the writing – see the subsections below.

In some routines, what are called MMR, MMI and MMC above are called MMRLR, MMRLI and MMRLCH respectively.

Example. Look in the code for the main program MKHAR. You will see the following call to GPBNF1 which determines MMRLR (safe number of reals).

```
call GPBNF1('mmrlr', mmrlr, errmes)
if( errmes(1:1).eq.'E') goto 15000
```

Then you can see later a call to the PUT7RE routine which has MMRLR passed to it. [PUT7RE writes an array to a header on a Header Array file – see section 7.1.]

Example. You can see a call to routine GPBNF1 in subroutine ReadWriteOneHeader in example EXRES1.F90 (see section 3). The value of MMR returned from this is passed to the call to routine PUT7RE which writes the array to the file.

6.8 Unit Numbers – Routines GPUNI2 and GPUNIT

GEMPACK programs do not hard-wire unit numbers since different numbers may be used conventionally by different compilers. Instead getting and freeing of unit numbers is done via routines GPUNI2 and GPUNIT.

To get a new unit number for use, it is best to call routine GPUNI2. That returns a new unit number which you can use for attaching a file to. The code looks as follows:

```
undesc=<string>
call GPUNI2('new', <unit-number>, undesc, errmes)
```

Here the UNDESC string may be shown in error messages. For example, if there is an error reading from the file attached to unit number 26 and if that unit number was associated with

SLNUN in the call to GPUNI2, the error message will mention SLNUN as well as number 26. Then you will know what file is attached to this unit.⁶

When you have finished with the unit number, you should return it to the store of available unit numbers using a call to routine GPUNIT of the form

```
call GPUNIT('end', <unit-number>, errmsg)
```

Example. You can see examples of calls to GPUNI2 in the example programs EXWHA1.FOR (section) and EXRES1.F90 (section). But you won't see any calls to free the unit numbers in those programs since the unit numbers obtained via the GPUNI2 calls are needed for the whole program.

Example. You can see a call to GPUNIT to free a unit number in SAGEM – see the code

```
call GPUNIT( 'end', xcmfun, errmsg)
```

which returns unit number XCMFUN to the pool of available unit numbers for use.

⁶ Note that UNDESC is of length LLUNDESC which is declared in module GPBAS1.

7 Reading and Writing Header Array Files

Each header on a Header Array file can contain real, integer or character data. The associated data types are

- **RE** (7 dimensional, contains set and element labelling), **RL** (7 dimensional, no set and element labelling) or **2R** (2 dimensional, no set and element labelling) for arrays of reals,
- **2I** for a matrix of integers (no set and element labelling)
- **1C** for a vector of character strings.

In the sections below we describe the main routines for writing and reading real arrays.

Provided you use the GEMPACK subroutines for opening, closing, reading and writing Header Array files, you do not need to know the details of the structure of a Header Array. The reading routines use the Header as a label to find the relevant array for you.

7.1 Writing a Real Array

7.1.1 Writing a Real Array Without Set and Element Labelling

Subroutine PUT7R is the main routine used for writing a real array when you do not want set and element labelling. Its calling sequence is:

```
CALL PUT7R( UNIT, HEADER, ARR,  
*   DIM1, DIM2, DIM3, DIM4, DIM5, DIM6, DIM7,  
*   ST1, ST2, ST3, ST4, ST5, ST6, ST7,  
*   N1, N2, N3, N4, N5, N6, N7,  
*   LNAME, MMRECL, report,  
*   ERRMES)
```

Here all arguments are intent(in) [that is, are input, not altered] except for ERRMES.

UNIT (integer) is the logical unit number of the file are writing to.

HEADER (character string) is the header are writing to.

ARR(DIM1,DIM2,DIM3,DIM4,DIM5,DIM6,DIM7) is the real array are writing from. DIM1 to DIM7 (integers) are usually the sizes of the array as it is declared in the main program or calling routine.

Are writing the part of the array starting from positions ST1,ST2 up to ST7 (integers). [Often all of these are equal to 1.]

Are writing the array of size N1xN2xN3xN4xN5xN6xN7 (integers).

LNAME (character string) is the long name are writing on the file.

MMRECL (integer) is the maximum number of reals to write in one physical record (see section 6.7).

REPORT is a logical variable which tells whether or not the writing is reported to the screen (or terminal),

ERRMES is the usual error character string.

Subroutine PUT7R writes an array with data type RL.

Example. See the top part of the call to PUT7RE in routine WriteData in example program EXWHA1.FOR. (The bottom part of that call deals with the set and element labelling – see section .)

7.1.2 Writing a Real Array With Set and Element Labelling

Subroutine PUT7RE is the main routine you will use when you want to write a real array with set and element labelling. Its calling sequence is:

```
CALL PUT7RE( UNIT, HEADER, ARR,  
*   DIM1, DIM2, DIM3, DIM4, DIM5, DIM6, DIM7,  
*   ST1, ST2, ST3, ST4, ST5, ST6, ST7,  
*   N1, N2, N3, N4, N5, N6, N7,  
*   writse,  
*   acdmkn, acdim, csname,  
*   stknow, ddset, stname, stelst, elno, elad,  
*   mmel, stel,  
*   LNAME, MMRECL, report,  
*   ERRMES)
```

Here all arguments are intent(in) [that is, are input, not altered] except for ERRMES.

The first arguments (down to N7) and the last four are the same as in the call to routine PUT7R – see section 7.1. The other arguments that include the information necessary for set and element labelling are as follows.

WRITSE tells if are to write set and element labelling information. (If WRITSE is false, the other arguments described below are ignored.)

ACDMKN (logical) tells if the actual dimension of the array is known.

ACDIM (integer) is the actual dimension of the array (irrelevant if ACDMKN is false).

CSNAME (character string) is the name of the Coefficient usually associated with this array.

STKNOW (logical) tells if all required information about the sets and elements associated with this array are known. If STKNOW is true, this means that ACDIM is known (that is, ACDMKN is true) and that STNAME(1..ACDIM), STELST(1..ACDIM), ELNO(1..ACDIM), ELAD(1..ACDIM) and the relevant parts of STEL are all known.

DDSET (integer) is the dimension of the arrays STNAME, STELST, ELNO, ELAD as passed in – it is usually 7.

STNAME(DDSET) is an array of character strings containing the names of the associated sets. The strings in this array must be of length 12 (LLST).

STELST(DDSET) is an array of characters (each of length 1) which tells if the elements of the sets are known 'k' or unknown 'u' or if the relevant set is actually a single element 'e'.

ELNO(DDSET) is an array of integers which tell the element number in the relevant sets for all cases where this dimension is an element (that is, for which STELST(I)='e').

ELAD(DDSET) is an array of integers. ELAD(I) tells the starting position in array STEL of the elements of set number I. [ELAD(I) is irrelevant if STELST(I)='u'.]

MMEL (integer) is the dimension of the array STEL.

STEL(MMEL) is an array of character strings which contain the elements of the different sets. The strings in this array must be of length 12 (LLSTEL).

Subroutine PUT7RE writes an array with data type RE if logical variable WRITSE is true, otherwise it writes an array with data type RL.

Example. See how the values in these arrays are set up in routine WriteData in example program EXWHA1.FOR. For example, STEL can contain the elements in any order. The values of ELAD(I) determine which elements belong to which sets.

7.2 Reading a Real Array

How you read a real array on a Header Array file depends on

- whether or not the size of the array is known in advance. If so you will probably want to check that the size of the array found at the header is the same as you expect.
- whether you wish to read the data into a real vector or a real 7-dimensional array in memory, and
- whether or not you want to read any set and element labelling on the file.

We describe these different cases in the subsections below.

The routines FGTR7C, GTRVAD, FGR7CE and GRVADE described below can all read arrays of any real type – that is, can read from arrays of types 2R, RL and RE.

Each of these subroutines will find the Header on the file. You do not need to find the position of the Header. After reading the array from the Header, the position on the file is just after the Header and its associated array. If you then want to read another Header, it will search through the Headers till the end of the file is reached. If the Header is still not found, the file is rewound at this stage and it will start at the beginning of the file and search through the file till the Header is found. If the Header is not found anywhere on the file, an error message will be written, and ERRMES will be set. After calling these subroutines you should always test ERRMES and use the standard error handling as described in section 6.1.

7.2.1 Reading, Not Set and Element Labelling, Checking Sizes

Here the relevant routine is FGTR7C which reads into a real 7-dimensional array from a Header Array file. FGTR7C does not read the set and element labelling even if it is present.

The **calling sequence for FGTR7C** is:

```
CALL FGTR7C( UNIT, HEADER, ARR,  
*   DIM1, DIM2, DIM3, DIM4, DIM5, DIM6, DIM7,  
*   ST1, ST2, ST3, ST4, ST5, ST6, ST7,  
*   N1, N2, N3, N4, N5, N6, N7,  
*   REPORT,  
*   LNAME,  
*   DWORK, WORK, WORKST,  
*   ERRMES)
```

Here UNIT, HEADER, DIM1 to DIM7, ST1 to ST7, N1 to N7, REPORT, DWORK and WORKST are all intent(in), the relevant part of ARR(DIM1,...,DIM7) is output, LNAME is intent(out) and WORK is a real work array.

UNIT (integer) is the logical unit number of the file you are reading from.

HEADER (character string) is the header you are reading from.

ARR(DIM1,DIM2,DIM3,DIM4,DIM5,DIM6,DIM7) is the real array you are reading into. DIM1 to DIM7 (integers) are usually the sizes of the array as it is declared in the main program or calling routine.

Are reading into the part of the array starting from positions ST1,ST2 up to ST7 (integers). [Often all of these are equal to 1.]

Are reading an array of size N1xN2xN3xN4xN5xN6xN7 (integers).

LNAME (character string) is the long name as read from the file.

REPORT is a logical variable which tells whether or not the writing is reported to the screen (or terminal).

DWORK (integer) is the dimension of the array WORK.

WORK(DWORK) is a real work array.⁷ Values may be put into it starting in position WORKST (integer). DWORK must be at least as large as (WORKST-1)+N1*N2*N3*N4*N5*N6*N7.

WORKST is an integer.

ERRMES is the usual error character string.

If you want to read into an array which is declared as a real vector (instead of a 7-dimensional array), you can do so with the call

```
CALL FGTR7C( UNIT, HEADER, ARR,
*   N1, N2, N3, N4, N5, N6, N7,
*   1, 1, 1, 1, 1, 1, 1,
*   N1, N2, N3, N4, N5, N6, N7,
*   REPORT,
*   LNAME,
*   DWORK, WORK, WORKST,
*   ERRMES)
```

This pretends that the real vector is declared as an N1xN2xN3xN4xN5xN6xN7 array. Of course you would need to check that the dimension of the array as declared is at least as large as product of N1*N2*N3*N4*N5*N6*N7.

7.2.2 Reading, Including Set and Element Labelling, Checking Sizes

Here the relevant subroutine is FGR7CE which reads into a real 7-dimensional array.

The **calling sequence for FGR7CE** is:

```
CALL FGR7CE( UNIT, HEADER, ARR,
*   DIM1, DIM2, DIM3, DIM4, DIM5, DIM6, DIM7,
*   ST1, ST2, ST3, ST4, ST5, ST6, ST7,
*   N1, N2, N3, N4, N5, N6, N7,
*   REPORT,
*   LNAME,
*   readse,
*   acdmkn, acdim, csname,
*   stknow, ddset, rdstnm, stno, rdelst, elno, rdelad,
*   mmel, dname2, stel,
```

⁷ This array is not always needed – see the code of routine FGR7CE and the variable called PERFM there. You can see that the routines FGTR7C and FGR7CE were written when we were conforming to the Fortran 77 standard for GEMPACK. Had we been conforming to the Fortran 90 standard, we would have made WORK a local allocatable array and only allocated memory to it when it is needed.

* DWORK, WORK, WORKST,
* ERRMES)

The arguments common to routine FGTR7C have the same meaning as there (see section 7.2.1).

READSE (logical, intent(in)) tells whether or not to read the set and element labelling (if present).

ACDMKN (logical, intent(out)) tells if the actual dimension of the array is known.

ACDIM (integer, intent(out)) is the actual dimension of the array (irrelevant if ACDMKN is false).

CSNAME (character string, intent(out)) is the name of the Coefficient usually associated with this array.

STKNOW (logical, intent(out)) tells if the sets and elements associated with this array are known. If STKNOW is true, this means that ACDIM is known (that is, ACDMKN is true) and that STNAME(1..ACDIM), STELST(1..ACDIM), ELNO(1..ACDIM), ELAD(1..ACDIM) and the relevant parts of STEL are all known. If STKNOW is false, you cannot use the values returned in STNAME, STELST, ELNO, ELAD and STEL. [Basically the set and element information is only useable if both ACDMKN and STKNOW are true.]

DDSET (integer) is the dimension of the arrays RDSTNM, RDELST, ELNO, RDELAD as passed in – it is usually 7.

RDSTNM(DDSET) is an array of character strings, intent(out), which holds the names of the associated sets. The set associated with argument number I is RDSTNM(I) [I=1,..,ACDIM]. The strings in this array must be of length 12 (LLST).

STNO(DDSET) is an array of integers, intent(out), which holds the set numbers of the associated sets. This array points into the RDSTNM array. For example, the set associated with the 3rd argument of the array read is RDSNTM(STNO(3)). See the COMxREGxREG example in section 7.2.3 for the reason why both RDSTNM and STNO arrays are returned.

RDELST(DDSET) is an array of characters, intent(out), (each of length 1) which tells if the elements of the sets are known 'k' or unknown 'u' or if the relevant set is actually a single element 'e'.

ELNO(DDSET) is an array of integers, intent(out), which tell the element number in the relevant sets for all cases where this dimension is an element (that is, for which RDELST(I)='e'). [If RDELST(I) is not 'e', ELNO(I) is set equal to 0 to indicate at set rather than an element.]

RDELAD(DDSET) is an array of integers, intent(out). RDELAD(I) tells the starting position in array STEL of the elements of the set associated with argument number I. The set number associated with argument number I is STNO(I) and the set name is RDSTNM(STNO(I)). [RDELAD(I) is irrelevant if RDELST(I)='u'.]

MMEL (integer, intent(in)) is the dimension of the array STEL.

DNAME2 (character string, intent(in)) is the name of MMEL in the main program.

STEL(MMEL) is an array of character strings, intent(out), which contain the elements of the different sets. The strings in this array must be of length 12 (LLSTEL).

See section 7.2.3 for examples of the values relating to set and element information returned by this routine.

7.2.3 Examples of Set and Element Labelling Information Returned

The examples below help in understanding the significance of the values returned by routines FGR7CE (see section 7.2.2) and GRVADE (see section 7.2.4) for the set and element labelling information.

Example 1. Suppose that you have just read the values of a real array of size COMxIND – for example, intermediate inputs of domestic commodity C into industry I. Suppose that the set COM has elements com1-com3 and that the set IND has elements i1-i4. Then the values returned by FGR7CE or GRVADE would be

```
ACDIM=2, RDSTNM(1)='COM', RDSTNM(2)='IND',  
STNO(1)=1, STNO(2)=2, RDELST(1)=RDELST(2)='k',  
ELNO(1)=ELNO(2)=0, RDELAD(1)=1, RDELAD(2)=4,  
STEL(1)='com1', STEL(2)='com2', STEL(3)='com3', STEL(4)='ind1', STEL(5)='ind2',  
STEL(6)='ind3', STEL(7)='ind4'.
```

Example 2. Suppose that you have just read the values of a real array of size COMxREGxREG – for example, exports of commodity C from region R to region S. Suppose that set COM has elements com1-com3 and that set REG has elements r1 and r2. Then the values returned by FGR7CE or GRVADE would be

```
ACDIM=3, RDSTNM(1)='COM', RDSTNM(2)='REG', RDSTNM(3)='REG',  
STNO(1)=1, STNO(2)=2, STNO(3)=-2, RDELST(1)=RDELST(2)=RDELST(3)='k',  
ELNO(1)=ELNO(2)=ELNO(3)=0, RDELAD(1)=1, RDELAD(2)=4, RDELAD(3)=2,  
STEL(1)='com1', STEL(2)='com2', STEL(3)='com3', STEL(4)='r1', STEL(5)='r2'.  
[Note that STNO(2)=STNO(3)=2 because the sets associated with the second and third arguments are the same, namely the set RDSTNM(2)='REG'.]8
```

Example 3. Suppose that you have just read the values of the COMx"imp"xIND part of the array BAS1(C,S,I) which represents the intermediate inputs of commodity C from source S into industry I.⁹ Suppose that COM and IND have elements as in the examples above and that the set SRC has the elements dom and imp. Then the values returned by FGR7CE or GRVADE would be

```
ACDIM=3, RDSTNM(1)='COM', RDSTNM(2)='SRC', RDSTNM(3)='REG',  
STNO(1)=1, STNO(2)=2, STNO(3)=-3, RDELST(1)='k', RDELST(2)='e', RDELST(3)='k',  
ELNO(1)=0, ELNO(2)=2, ELNO(3)=0, RDELAD(1)=1, RDELAD(2)=4, RDELAD(3)=5,  
STEL(1)='com1', STEL(2)='com2', STEL(3)='com3', STEL(4)='imp', STEL(5)='r1',  
STEL(6)='r2'.  
[Note that only the relevant element imp of the set SRC is available in STEL (the first element dom is not in STEL). RDELST(2)='e', ELNO(2)=2 and RDELAD(2)=4 are what tell that we have read the "imp" slice of the full COMxSRCxIND BAS1 array.]
```

7.2.4 Reading, Not Set and Element Labelling, Returning Sizes

Here the relevant routine is GTRVAD which reads into a real vector.

The **calling sequence for GTRVAD** is:

⁸ STNO(I) points to the first set in the RDSTNM array which is equal to RDSTNM(I) – see the code of GEMPACK routine RDHSE which actually reads the set and element labelling information and assigns values to the STNO array. The elements of the different sets involved are stored on the file with the data. One reason for returning RDSTNM and STNO arrays is to avoid storing the elements of any set which is a repeated argument (for example, REG in COMxREGxREG) twice.

⁹ This is the data held at header F002 in the Monash FID data file (although there the sets COM and IND are larger than in the example in the text).

```

      CALL GTRVAD( UNIT, HEADER,
*      ARRAY, DIMARR, DNAME, ARRST,
*      NELS, MAXDIM, LNAME,
*      REPORT,
*      ERRMES)

```

The arguments of GTRVAD are:

UNIT, HEADER, LNAME, REPORT and ERRMES are as in FGTR7C.

DIMARR (integer, intent(in)) is the dimension of the array ARRAY are reading into.

DNAME (character string, intent(in)) is the name of DIMARR in the main program. [For example, if are passing an array declared as " REAL :: RealVec(DimRealVec)" in the main program, DNAME should be set equal to 'DimRealVec'.]

MAXDIM (integer, intent(in)) is the dimension of arrays ARRST and NELS. MAXDIM is usually 7 (or larger).

ARRST (integer, intent(in)) is the first position in ARRAY(DIMARR) are reading into. Often ARRST is equal to 1. [For example, if are reading a 3x4x2x1x1x1x1 array and if ARRST=10, then will read its values into positions 10 to 33 of ARRAY since are reading an array of size 24.]

NELS(MAXDIM) (integer, intent(out)) is the array indicating the size of the array read in. An array of size

NELS(1)xNELS(2)xNELS(3)xNELS(4)xNELS(5)xNELS(6)xNELS(7)
is read into ARRAY starting at position ARRST.

ARRAY (intent(out)) is a real vector.

As ever with Fortran, when an array is stored in a vector, the values are stored in column order, which means that the first dimension varies fastest and the last dimension varies slowest. For example, suppose that we have read a 3x4x2x1x1x1x1 array and suppose ARRST=10. Then

- position 10 of ARRAY holds entry in position (1,1,1,1,1,1) of the array,
- position 11 of ARRAY holds entry in position (2,1,1,1,1,1) of the array,
- position 13 of ARRAY holds entry in position (1,2,1,1,1,1) of the array,
- position 23 of ARRAY holds entry in position (3,2,4,1,1,1) of the array.

7.2.5 Reading, Including Set and Element Labelling, Returning Sizes

Here the relevant routine is GRVADE which reads into a real vector.

The **calling sequence for GRVADE** is:

```

      CALL GRVADE( UNIT, HEADER,
*      ARRAY, DIMARR, DNAME, ARRST,
*      NELS, MAXDIM, LNAME,
*      readse,
*      acdmkn, acdim, csname,
*      stknow, ddset, rdstnm, stno, rdelst, elno, rdelad,
*      mmel, dname2, stel,
*      REPORT,
*      ERRMES)

```

The arguments common to routine GTRVAD have the same meaning as there (see section 7.2.3). These are the arguments associated with the real array ARRAY.

The arguments common to routine FGR7CE (see section 7.2.2) have the same meaning as there. These are the arguments associated with the set and element labelling information.

See section 7.2.3 for examples of the values relating to set and element information returned by this routine.

7.3 Opening and Closing Header Array Files

7.3.1 Opening a Header Array File

Exactly which GEMPACK subroutine to use depends on whether or not the name has already been specified. If you want the user to specify the name and then to open the file, you can use routine OPFILE. Its calling sequence is

```
CALL OPFILE( UNIT, STATUS, FTYPE,
*          FNAME, OPENOK,
*          ERRMES )
```

Here arguments UNIT, STATUS and FTYPE are intent(in) [that is, are input, not altered] while arguments FNAME and OPENOK are intent(out) [that is, are output from this routine].

UNIT (integer) is the logical unit number that the file will be opened on.

STATUS (character string) is either 'old' (if the file must already exist) or 'new' (if a new one is to be created).

FTYPE (character string) is 'header' for a Header Array file. [Other possible values are 'formatted', 'unformatted' or 'print'.]

FNAME (character string) is the name of the file (as read from the terminal).

OPENOK (logical) tells whether or not the file has been opened ok. The value of this must be tested by the calling routine. ERRMES is not set fatal just because OPENOK is false.

ERRMES is the usual error character string.

Example. See the call to OPFILE to open the new Header Array file in example program EXWHA1.FOR. Note that prompt (via call to GPTO) before it.

7.3.2 Closing an Old Header Array File

An old Header Array file (that is, one which existed before the program was run) can be closed via a call to routine CLSEQ. The calling sequence is

```
CALL CLSEQ(UNIT, STATUS, ERRMES)
```

Here the first two arguments are intent(in) [that is, are input, not altered].

UNIT (integer) is the logical unit number that the file was opened on.

STATUS (character string) is either 'keep' (if the file is to be kept) or 'delete' (if the file is to be deleted). Usually STATUS='keep' when closing an old file.

7.3.3 Closing a New Header Array File

A new Header Array file (that is, one which has been created by the program) should be closed via a call to routine CLNWHF. The calling sequence is

```
CALL CLNWHF( UNIT, STATUS, ERRMES )
```

Here the first two arguments are intent(in) [that is, are input, not altered].

UNIT (integer) is the logical unit number that the file was opened on.

STATUS (character string) is either 'keep' (if the file is to be kept) or 'delete' (if the file is to be deleted). Usually STATUS='keep' when closing a new file.

Before closing the file, this routine CLNWHF checks to see if there are any duplicate headers on the file. If so, a fatal error is set. This is why you should call routine CLNWHF rather than the alternative closing routine CLSEQ (see section 7.3.2) which does not check for duplicate headers.

Example. See the call to CLNWHF to close the new Header Array file in example program EXWHA1.FOR.

7.3.4 Writing Creation Information and History

If you want to write the GEMPACK-style Creation Information on a new file, you can do so via a call to routine WRCREF. Its calling sequence is

```
CALL WRCREF( UNIT, PROGNV, FTYPE, ERRMES )
```

where the arguments are

UNIT (integer) is the logical unit number that the file was opened on.

PROGNV is the program name and version (as usually declared in the main program – see MODELPRG).

FTYPE (character string) is 'header' for a Header Array file. [Other possible values are 'formatted', 'unformatted' or 'print'.]

If you write Creation Information on the file, this will be echoed when another GEMPACK program (or one you write) opens the file. This echoing looks like:

```
! This file was created at 11:34:44 on 20-MAR-96 by the program
! <GEMSIM Version 1.3 February 1996>
! which accesses some of the routines in the GEMPACK software release
! <GEMPACK Release 5.1.6+ March 1996>
```

8 Allocating Memory for Arrays

When you write Fortran 90 code, you must manage the memory needed for the various arrays. If you are not going to rewrite the program every time the arrays change size, this is done by allocating memory for arrays and then deallocating that memory when the arrays are no longer needed.

Deciding how much memory to allocate is an issue.

In GEMPACK programs, we often do a preliminary read of part of a file in order to find out how much memory we need to allocate. You can see examples of this in EXWHA1.FOR and EXRES1.F90.

- In EXWHA1.FOR, there is a call to routine GetSizes which tells how large the input and output arrays need to be. That is followed by an ALLOCATE statement to allocate memory for the various arrays. The memory for these arrays is deallocated near the end of the main program.
- In EXRES1.F90, you can see memory being allocated and deallocated each time an array is read in subroutine ReadWriteOneHeader. There the routine HAHMD1 is called to ascertain how many real numbers are at the relevant header. Then the arrays are allocated. They are deallocated just before the subroutine returns.

We use variable ASTAT to hold the STAT value returned by ALLOCATE and DEALLOCATE statements. We use statement numbers 24200 and 24300 to trap for allocate and deallocate errors respectively.

9 Where to Find the GEMPACK Subroutines

You can find the subroutine and modules supplied with a Source-code Version of GEMPACK by looking inside the various .ZIP files in the SUBS, TABLO, MODULES and INCLUDES subdirectories. The GEMPACK main programs (for example, mkhar.for) are in the main GEMPACK directory (usually C:\GP).

You can put all the different files into a SOURCE subdirectory by running MKSOURCE.BAT from the main GEMPACK directory. Then you can find any routine you want in the SOURCE directory.

10 Compiling and Linking Your Programs

- If you write a fixed-format program you should give it suffix .FOR. To compile and link <myprog>.for, use the DOS command

```
ltnomod <myprog>
```

- If you write a free form program you should give it suffix .F90. To compile and link <myprog>.f90, use the DOS command

```
ltnfnmod <myprog>
```

In either case, this finds the relevant GEMPACK module files (in subdirectory MODULES) and links your program to the various GEMPACK subroutine libraries (including GPA.LIB and GPB.LIB).

10.1 Using the LF95 Compiler for Code Development

We use LF95 (rather than LF90) when we are developing GEMPACK programs and routines. We have found it has the better debugging features.

In development mode, you will probably want maximum checking to be done by the compiler and associated run-time system. We do this by running DEBFIG.BAT in the main GEMPACK directory. That sets various compiler checking options in the LF95.FIG file (in the main GEMPACK directory). That LF95.FIG file controls compilation done using the ltgnomod and ltgfnmod commands (see section 10).

For example, the various debugging options we set with LF95 report errors

- if a variable is used but not declared.
- if a subroutine is called with a different number of arguments (or different types of arguments) from its declaration.
- if a array subscript goes out of range.
- if a subroutine appears to use a larger part of an array than the size of the array in the calling routine.

The compiler traps the first of those above while the other errors are only reported when you run the program.

When you are ready to build an executable image for production use, you can get back the "fast" compiler options by running LTGFST.BAT in the main GEMPACK directory. [This removes the checking parts of LF95.FIG.]

11 References

Harrison, W.J. and K.R. Pearson (1996), 'Computing Solutions for Large General Equilibrium Models Using GEMPACK', *Computational Economics*, vol. 9, pp.83-127.