**SUN/150.10**

Starlink Project
Starlink User Note 150.10

D.S. Berry

20th October 2012

---

# GRP
# Routines for Managing Groups of Objects
# Version 3.7
# Programmer's Manual

---

## Abstract

GRP provides facilities for handling groups of objects which can be described by character strings (e.g. file names, astronomical objects, numerical values, etc). Facilities are provided for storing and retrieving such character strings within groups, and for passing groups of character strings between applications by means of a text file. It also allows the creation of groups in which each element is a modified version of an element in another group.

# Contents

# 1 Introduction

Applications often have to manage groups of strings in which each string represents some object. Typical examples of the contents of such groups may be:

- Names of astronomical objects.
- File names.
- Lists of wavelengths of spectral lines.

The GRP package provides facilities for managing the storage and retrieval of strings within such groups. Applications may explicitly specify strings to be stored within a group, or instead may specify the name of a text file from which to read such strings. The contents of a group may be written out to a text file, providing an easy means of passing groups of objects between applications.

## 1.1 Facilities Provided by GRP

The concept of a group within GRP may be compared with an "array" structure within a high level programming language such as Fortran. Arrays are used for storing many values in a single object, with each value associated with an integer "index". A value can be stored in, or retrieved from, a particular element of an array by specifying the index for that element. The facilities provided by the GRP package are similar. GRP allows character strings to be stored in, or retrieved from, one of several "groups" (in this sense, a "group" is the GRP equivalent of a character array). Each element within the group has an associated index, and different groups are distinguished by different "identifiers" (similar to the way that different arrays are distinguished by having different names).

GRP also provides the following features not available through the use of Fortran arrays:

(1) In order to store values in a normal character array, you must assign an explicit character string to each element of the array. The GRP package provides similar facilities for storing explicit strings, but also provides facilities for reading the values to be stored from a text file. This is known as *indirection*; instead of providing a set of strings to be stored, you provide the name of a file which contains the strings to to be stored. For instance, if you were prompted for a list of data files to be processed, you could then respond either with the explicit name of each data file, or with the name of a text file containing a list of the names of the data files.

(2) An alternative method for specifying values to be stored in a group is by modification of the values already stored in *another* group. For instance, if values are to be assigned to elements 1 to 3, the actual strings stored would be obtained by taking the values stored in elements 1 to 3 of another group and applying some editing to them. The same editing is used for each element; it can include the addition of a suffix and/or a prefix, and the substitution of one sub-string with another. A typical use of this facility could be to specify a set of output files by describing the editing and the names of the corresponding input files. Thus, if a group describing the input files contained the three strings:

```
OBJ1.DAT
OBJ2.DAT
OBJ3.DAT
```

you might specify the output files using a string such as:

```
A_*|DAT|TXT|
```

This would cause the addition of the prefix "**A_**" and substitution of "**TXT**" for "**DAT**", resulting in the output file names:

```
A_OBJ1.TXT
A_OBJ2.TXT
A_OBJ3.TXT
```

(3) In the above example, a group of values was obtained by copying them from another group, and then applying some specified editing. This technique of editing values can be combined with other ways of specifying the original values. For instance, to apply the same editing as above to each of the values stored in the text file `OBJECTS.LIS` the following response could be given:

```
A_{^OBJECTS.LIS}|DAT|TXT|
```

If the same editing is to be applied to a list of literal values typed in directly at the keyboard, then a response such as the following could be given:

```
A_{OBJ1.DAT,OBJ2.DAT,OBJ3.DAT}|DAT|TXT|
```

(4) Groups have dynamic size (i.e. they expand in size as required to make room for new values), which is often more convenient than the fixed size of an array specified in its declaration. Thus if an application wants to process many data files, and chooses to store the file names in a GRP group, then no limit is imposed on the number of files which the user may specify.

(5) The contents of a group may be written out to a text file using a single subroutine call. This, together with the ability to read a group's contents from a text file, provides a convenient means of passing groups of objects between applications. For instance, an application may produce a text file holding the names of all the output data files it has created. A later application can then read this text file to obtain the names of the data files which it is to process.

(6) Elements within a group can be copied in a single call to another group. Duplicate names may also be purged from a group in a single call.

(7) GRP stores information about how each value within a group group was obtained (i.e. whether it was given explicitly, or by indirection, or by modification). The names of indirection files are stored, as are the identifiers for groups used as the basis for "modified" elements, and all this information is available to the calling application.

GRP is a general purpose library which makes no attempt to attach any particular meaning or properties to the strings stored in a group. It is expected that other, more specialized libraries will be written which use GRP to handle specific types of strings (eg coordinate values, names of data files, etc). Such packages will provide additional features to handle the objects stored by GRP (eg the creation, opening and closing of data files).

## 1.2 An Example of a Simple GRP Application

The facilities of GRP are particularly useful for processing lists of text strings provided in response to a prompt. The user of an application can specify the strings literally, or can specify the name of a text file containing the strings, or can specify the editing to be used to derive them by modification of the strings stored somewhere else.

### 1.2.1 Example Code

Here is a simple example of the use of GRP which illustrates the ideas of indirection and modification. In this example, each stored string corresponds to the name of a file but obviously an application could apply other interpretations. The user is prompted for a set of input file names and then prompted again for a set of output file names. Each input file is processed in some way (by routine PROC) to produce the corresponding output file. The source code that follows is not intended to provide all the information necessary to write GRP applications, but simply to give a feeling for the way GRP works:

```
      SUBROUTINE GRP_TEST( STATUS )                              [1]
      IMPLICIT NONE

*  Include definitions of global constants.
      INCLUDE 'SAE_PAR'                                          [2]
      INCLUDE 'GRP_PAR'                                          [3]

*  Declare local variables.
      INTEGER STATUS, IGRP1, IGRP2, SIZE1, SIZE2, ADDED, I
      CHARACTER*(GRP__SZNAM) INFIL, OUTFIL
      LOGICAL FLAG

*  Check inherited global status.
      IF ( STATUS .NE. SAI__OK ) RETURN                          [4]

*  Create a new (empty) group to contain the names of the
*  input files.
      CALL GRP_NEW( 'Input files', IGRP1, STATUS )               [5]

*  Prompt the user for a group of input file names and place
*  them in the group just created.
      CALL GRP_GROUP( 'IN_FILES', GRP__NOID, IGRP1, SIZE1,       [6]
     :                ADDED, FLAG, STATUS )

*  Create a second group to hold output file names.
      CALL GRP_NEW( 'Output files', IGRP2, STATUS )              [7]

*  Prompt the user for a group of output file names, giving
*  the chance to specify them by modification of the input
```

```
*  file names. Place the output file names in the new group
*  just created.
      CALL GRP_GROUP( 'OUT_FILES', IGRP1, IGRP2, SIZE2,          [8]
     :                ADDED, FLAG, STATUS )

*  Report an error and abort if the number of output files
*  does not equal the number of input files.
      IF( SIZE2 .NE. SIZE1 .AND. STATUS .EQ. SAI__OK ) THEN      [9]
         STATUS = SAI__ERROR
         CALL ERR_REP( 'GRP_TEST_ERR1',
     :           'Incorrect number of output files specified',
     :           STATUS )
         GO TO 999
      END IF

*  Loop round each input file.
      DO I = 1, SIZE1                                           [10]

*  Retrieve the input file name with index given by I.
         CALL GRP_GET( IGRP1, I, 1, INFIL, STATUS )             [11]

*  Retrieve the output file name with index given by I.
         CALL GRP_GET( IGRP2, I, 1, OUTFIL, STATUS )

*  Process the data.
         CALL PROC( INFIL, OUTFIL, STATUS )                     [12]

*  Do the next input file.
      END DO

*  Delete the groups created by this application.
 999  CONTINUE
      CALL GRP_DELET( IGRP1, STATUS )                           [13]
      CALL GRP_DELET( IGRP2, STATUS )

      END
```

**Programming notes:**

(1) The example is actually an ADAM A-task, and so consists of a subroutine with a single argument giving the inherited status value. See SUN/101 for further details about writing ADAM A-tasks. A "stand-alone" version of the GRP package is available which can be used with non-ADAM applications.

(2) The first INCLUDE statement is used to define standard "symbolic constants", such as the values SAI__OK and SAI__ERROR which are used in this routine. Starlink software makes widespread use of such constants, which should always be defined in this way rather than by using actual numerical values. The file SAE_PAR is almost always needed, and should be included as standard in every application.

(3) The second INCLUDE statement performs a similar function to the first, but defines symbolic constants which are specific to the GRP package. Such constants are recognizable by the fact that they start with the five characters "GRP__" (such as GRP__NOID and

GRP__SZNAM used in the above example). Note the double underscore "__" which distinguishes them from subroutine names.

(4) The value of the STATUS argument is checked. This is because the application uses the Starlink error handling strategy (see SUN/104), which requires that a subroutine should do nothing unless its STATUS argument is set to the value SAI__OK on entry. Here, we simply return without action if STATUS has the wrong value.

(5) An identifier for a new group is now obtained. The variable IGRP1 is returned holding an integer value which the GRP package uses to recognise the group just created. Initially, there are no names stored in the group. A string is stored which should be used to give a description of the type of objects stored within the group (in this case the string *"Input files"* is used).

(6) The user is now prompted for a value for the parameter IN_FILES, and replies with a string, which GRP splits up into separate elements, each element being either a literal file name or the name of a text file containing other file names. As there are no other groups defined at this point, it is not possible to specify the files names using "modification" (as described in item (2) in section 1.1). For this reason, the second argument (which would normally specify the group to use as the basis for modification) is given the value GRP__NOID. This is a special identifier value used to indicate a "null" group. The names supplied by the user are stored in the group created by the previous call to GRP_NEW, and the number of names is returned in argument SIZE1. Note, no permanent association exists between the *group* identified by IGRP1 and the *parameter* IN_FILES (which is one reason why GRP_GROUP is not called GRP_ASSOC). The parameter value may be cancelled (for instance using PAR_CANCL) without effecting the contents of the group.

(7) A second group is now created to hold the output file names. The two groups are distinguished by the fact that they have different identifiers (stored in IGRP1 and IGRP2).

(8) The user is prompted again, this time for a value for the parameter OUT_FILES and the names obtained are stored in the second group just created. Again the user can give literal files names and/or the names of text files holding other file names. Now that there are two groups, it is possible to use modification to specify the output files. The identifier for the group containing the input files names is given as the second argument of GRP_GROUP, telling the GRP system that *if* the user specifies output file names using modification (which may not be the case of course), then the output file names are to be derived by modifying the input file names stored in the first group.

(9) In this particular application it is deemed necessary to have equal numbers of input and output files, but GRP imposes no restrictions on the number of strings which can be supplied when responding to a prompt from GRP_GROUP. It is therefore necessary to check the that the two groups contain the same number of elements. A more sophisticated application might seek user intervention to determine how to proceed at this point (either by requesting extra output file names or by ignoring some). Note, if some other error has already been detected (as shown by STATUS having a value other than SAI__OK), then the check on the number of input and output files is irrelevant.

(10) Having stored the input file names in one group and the output file names in another, the application now loops through each pair of input and output file names in turn. An "index" I is used to distinguish between different elements within a group. The input file

name with index I is retrieved from the first group and the output file name with the same index is retrieved from the second group.

(11) Note, there is a limit to the size of the character string which can be stored in a GRP group. This size is given by the symbolic constant GRP__SZNAM.

(12) A routine is now called which uses the two file names; a typical routine may take data out of the input file, process it and store the results in the output file. The file handling itself would be done within the routine PROC. This example actually makes no assumptions about what the strings stored in the two groups represent (the descriptive strings stored when the two groups were created are of no significance in this application). Although, for clarity, it has been assumed that the strings correspond to file names, they could just as easily have been wavelengths, the names of astronomical objects, calender dates, or just about anything else.

(13) Finally, the groups created by this application are deleted. This is particularly important in applications which run as subroutines within a wider context (such as ADAM applications). There is a limited number of groups available, and if applications forget to delete the groups they have created, then the possibility of exceeding the limit then exists. Note, the groups should be deleted even if the application aborts because of an error, so the statement labelled 999 (to which a jump is made if an error is detected) comes *before* the calls to GRP_DELET.

### 1.2.2 Examples of Possible User Responses

If the example above was run, the user could respond in several ways to the prompts for parameters IN_FILES and OUT_FILES. The following paragraphs illustrate the use of indirection and modification in this context.

**Indirection.** When prompted for IN_FILES the user could reply with the following text:

```
IC_1575_RAW, IC_4320_RAW, ^NGC_OBJECTS.LIS
```

This would cause the GRP package to place the two strings `IC_1575_RAW` and `IC_4320_RAW` in two elements of the first group and then search for a file called `NGC_OBJECTS.LIS`. If this file contains the following two lines of text:

```
NGC_5128_RAW, NGC_2534_RAW
^OTHERS.LIS
```

then the strings `NGC_5128_RAW` and `NGC_2534_RAW` would be added to the same group, and a search made for the file `OTHERS.LIS`. If this file, in turn, contained the two lines:

```
NGC_1947_FLAT
NGC_3302_FLAT
```

then the final group would consist of the six strings:

```
IC_1575_RAW
IC_4320_RAW
NGC_5128_RAW
NGC_2534_RAW
NGC_1947_FLAT
NGC_3302_FLAT
```

This illustrates the use of indirection as a means of specifying the strings to be stored in a group, and shows it being combined with the specification of literal strings, and indirections being nested.

Strings stored in a text file can be edited "on the fly" before being stored in a group. For instance, the user could give the following response to a prompt for IN_FILES:

```
^NGC_OBJECTS.LIS|_RAW|_CAL|
```

This would cause the GRP package to read the values stored in text file `NGC_OBJECTS.LIS`, replacing all occurrences of the string "_RAW" with the string "_CAL". If the file `NGC_OBJECTS.LIS` contained the same values as before, then the editing would also be applied to the values stored in the file `OTHERS.LIS`.

**Modification.** As an example of the use of "modification", let's suppose that the user responds to the prompt for OUT_FILES with the string:

```
A_*2|RAW|FLAT|
```

This would cause the application to generate six strings, based on the six strings held in the first group (see programming note (8)). The names are generated as follows

(1) First, a copy of the six names in the first group is made and stored in the second group.

(2) Next, any occurrence of the string "RAW" within any of these names is replaced with the string "FLAT". This leaves the second group holding the names:

```
IC_1575_FLAT
IC_4320_FLAT
NGC_5128_FLAT
NGC_2534_FLAT
NGC_1947_FLAT
NGC_3302_FLAT
```

(3) Next, each name is substituted in turn for the "$*$" character in the string to the left of the first "|" character. Thus each name is prefixed by "A_" and suffixed by "2". The second group finally holds the names:

```
A_IC_1575_FLAT2
A_IC_4320_FLAT2
A_NGC_5128_FLAT2
A_NGC_2534_FLAT2
A_NGC_1947_FLAT2
A_NGC_3302_FLAT2
```

Modification can be combined with indirection and/or the specification of literal strings. For instance, the user could have replied to the prompt for OUT_FILES with the string:

```
NEW_FILE,A_*2|RAW|FLAT|,^LIST.DAT
```

This would have caused the second group to contain not only the six names described above, but also the additional names **NEW_FILE** and any names read from the text file LIST.DAT. In this case, the number of output files would have exceeded the number of input files and the check described in programming note (9) would fail.

# 2 Names

Each string stored within a group is referred to in the rest of this document as a *name*. The term is used purely for convenience and is not meant to imply anything about what the strings actually represent. In some cases the strings may easily be thought of as *names* (for instance, in the example in the previous section the strings were the *names* of files), but this may not always be so.

The maximum length of a name is given by the symbolic constant GRP__SZNAM (which is currently 255). Leading spaces, and spaces within names are significant, but trailing spaces are ignored. Completely blank names are allowed, and null names (i.e. names of zero length) are treated as blank names.

## 2.1 Case Sensitivity

Names are always *stored* in the form in which they are supplied. However, when names are *retrieved* from a group they will be converted to upper case if the group has been designated as a case insensitive group. When groups are first created they are considered to be case sensitive (i.e. names are retrieved from the group in the same form in which they were stored). They can be made case *in*-sensitive by calling routine GRP_SETCS. Any comparisons involving names held in case insensitive groups are performed without reference to case.

# 3 Groups and Group Identifiers

Applications can maintain information about many independent groups simultaneously. The maximum number of simultaneously active groups is currently set to 500 and is given by the symbolic constant GRP__MAXG. There is no limit on the number of names that a single group can contain (other than those imposed by system quotas).

The GRP system distinguishes between different groups by use of an identifier system. When a new group is created, it is assigned a "GRP identifier" or "group identifier". This value is used to specify which group is to be acted upon by subsequent GRP calls. The symbolic constant GRP__NOID can be given in certain places where a GRP identifier would normally be given, in order to specify a "null" group. Note, identifiers in the GRP package cannot be "cloned" to

produce alternative, independent channels for accessing the same group. If two integer variables contain identifiers for a single group, then the integer values stored in the two variables will always be equal.

# 4 Group Expressions

One of the most useful routines within GRP is GRP_GROUP. This routine appends names to the end of a previously created group using a "group expression" obtained from the environment via a named parameter which can be of any type. The routine GRP_GRPEX also performs this function, except that the group expression is provided by the calling application, rather than being obtained through the parameter system.

This section describes the syntax of group expressions.

## 4.1 Elements and Delimiters

Group expressions may contain several "delimiter" characters (usually a comma although this can be changed, see section 5.7) and the substrings delimited by these characters are referred to as "elements". If there are no delimiters in a group expression, then the group expression consists of a single element. For instance, the group expression:

```
NEW_FILE,A_*2|RAW|FLAT|,^LIST.DAT
```

consists of the three elements **NEW_FILE**, **A_∗2|RAW|FLAT|** and **^LIST.DAT**. Note, delimiter characters are ignored if they occur within matching "nesting characters" (see section 4.5). For instance, nesting prevents the group expression:

```
FLATFIELD(100:200,20:220),OBJECT
```

being split into three elements instead of two (i.e. the first comma does not act as a delimiter because it occurs within a nest formed by matching parentheses).

Each element of a group expression may be a literal name (eg **NEW_FILE** in the previous example), or an "indirection element" or a "modification element". An indirection element specifies a text file from which further names are to be read (eg **^LIST.DAT** in the previous example). A modification element specifies an existing group of names which are to be used as the basis for the new names (eg **A_∗2|RAW|FLAT|** in the previous example). These are described in more detail below.

## 4.2 Editing of Names

Each element in a group expression will give rise to one or more names (depending on whether the element consists of a literal name, an indirection element or a modification element). These names may be edited before being stored in a group by including certain "editing strings" within the text of the element. The general format of an element with editing strings included is:

```
prefix{kernel}suffix|old|new|
```

The *kernel* string can be a single element, or can be a full group expression. Processing of the element proceeds as follows:

(1) The kernel is first expanded to give a list of literal names. This may involve reading names from files, copying names from another group, etc, depending on the exact nature of the kernel. The characters which mark the start and end of the kernel are known as the opening and closing kernel delimiters. They are usually set to be "{" and "}", but can be changed if needed.

(2) Each name is checked to see if it contains the *old* string. If it does, all occurrences of the *old* string are replaced by the *new* string. The character which delimits the *old* and *new* strings is known as the "separator" character and is usually a "|" character, but can be changed if needed. This substitution will be case sensitive if the group to which the names are to be added has been designated as case sensitive (see section 2.1). If no substitutions are to be performed then the old and new strings, together with the three separator characters, should be omitted.

(3) The *prefix* string is added to the start of each name, and the *suffix* string is appended to the end of each name. Either or both of these strings may be null (i.e. of zero length).

The names which result from this processing are then added to a group. If there is no ambiguity about where the kernel starts and finishes (for instance if the prefix and suffix are both omitted, and the kernel consists of a single element) then the kernel does not need to be enclosed within kernel delimiters. The contents of the kernel can be any group expression. In particular, the kernel can contain other nested kernels with their own associated editing strings.

Let's look at some examples:

```
A_{TOM,DICK,HARRY}_B
```

This will give rise to the three names **A_TOM_B**, **A_DICK_B** and **A_HARRY_B**.

```
^FILE.LIS|_OLD|_NEW|
```

This will read names from the text file `FILE.LIS` (see the description of indirection elements below), and replace all occurrences of the string "**_OLD**" within the names with the string "**_NEW**".

```
WW,{A,B_{ONE,TWO,THREE}|T|Z|,C}KK|_Z|_Y|
```

This is a complex example and needs looking at carefully. Looking at it at the highest level, it can be thought of as:

```
WW,{kernel}KK|_Z|_Y|
```

where `kernel` is the group expression:

```
A,B_{ONE,TWO,THREE}|T|Z|,C
```

The first and third elements in this inner group expression are simple literal names and give rise to the two names **A** and **C**. The second element specifies that the three names **ONE**, **TWO** and **THREE** are to be edited by replacement of the letter T by the letter Z, and the addition of the prefix **B_**. After editing, these three names become **B_ONE**, **B_ZWO** and **B_ZHREE**. So the total group specified by this inner kernel is:

```
A
B_ONE
B_ZWO
B_ZHREE
C
```

We can now go back and look at the full group expression in the form:

```
WW,{kernel}KK|_Z|_Y|
```

The first element specifies the single name **WW**. The second element specifies that each of the names arising from the expansion of the inner kernel (i.e. the names listed above) should be edited by replacing **_Z** with **_Y**, and then appending the suffix **KK**. Thus the final group contains:

```
WW
AKK
B_ONEKK
B_YWOKK
B_YHREEKK
CKK
```

### 4.3 Indirection Elements

An indirection element consists of an "indirection character" (usually "^" (up arrow) although this can be changed, see section 5.7) followed by the name of a text file. For instance, the group expression:

```
^raw_data
```

would cause GRP to search for a file called `raw_data`.

The specified file is read to obtain further names to be added to the group. Each line in the file is processed as if it were a separate group expression, and so may contain any combination of literal names, modification elements or further indirection elements. It is thus possible to get several levels of indirection, in which a literal name is specified within a text file, which is itself specified within an indirection element contained within another text file, etc. GRP imposes a limit of 7 levels of indirection, primarily to safe-guard against "run-away" indirection which happens (for instance) when a file specifies *itself* within an indirection element.

Indirection elements are always considered to be case sensitive, even if the group has been designated case insensitive. This is because file names on certain operating systems (eg UNIX) are always considered case sensitive, and so problems would arise while accessing indirection files if GRP was to consider them case *in*sensitive.

The file name can contain shell meta-characters (references to environment variables for instance) which will be expanded before the file is used.

### 4.4 Modification Elements

A modification element causes GRP to generate a set of names by copying the names from another group. These new names can then be modified using the facilities for editing names described above. The application specifies which group is to be used as the basis for the new names. A special character (usually a "∗" character, but this can be changed if required) is used as a token to represent all the names in the basis group. Thus:

```
*|_DS|_BK|
```

would cause all the names in the basis group to be modified by replacing the string **_DS** with the string **_BK**. The basis names can also be modified by the addition of a prefix and suffix. Following the description of name editing given above, you may expect the format to be (for instance):

```
Hello_{*}_Goodbye
```

in which the token character takes on the role of the kernel. This does in fact work, but in this case the opening and closing kernel delimiters ("{" and "}") can be omitted because there is no ambiguity about where the kernel starts and finishes. Thus a simpler form would be:

```
Hello_*_Goodbye
```

The addition of a prefix and suffix can be combined with substitution as usual. For instance, the element:

```
A*B|C|D|
```

would cause all occurrences of the letter C within the names of the basis group to be replaced with D, followed by the addition of the prefix A and the suffix B.

If a "null" group is specified as the basis group (i.e. the group identifier is given as GRP__NOID), then there are no names on which to base the new names and the token character is treated as a literal name. That is, if the user gave the group expression

```
A_*2|RAW|FLAT|
```

and the application had specified a null group as the basis for modification elements, the the specified editing would be applied to the *literal name* "∗", resulting in the single literal name "**A**_∗**2**" being added to the group.

### 4.5 Nesting Within Group Expressions

There is sometimes a clash of interests to be resolved when deciding on the best choice for the character which delimits elements within a group expression. The default delimiter character is the comma, but this character can sometimes be useful *within* an element, for instance when specifying a set of indices. For instance, if the user gave the group expression:

```
A(1,2),B(3,10)
```

in which each element is a literal name corresponding to an array element, it would be wrong to split this up using the commas as delimiters into the four strings "**A(1**", "**2)**", "**B(3**" and "**10)**".

To get round this particular problem, GRP ignores delimiters which occur within matching "nesting characters". There are two nesting characters, the "open nest" character (usually set to "(") and the "close nest" character (usually set to ")"). Thus in the above example, the commas occurring within the parentheses would not be treated as delimiters, resulting in the group expression being split into the two elements **A(1,2)** and **B(3,10)**. The characters to use as the opening and closing nest characters may be set by the calling application (see section 5.7).

## 4.6 Flagging a Group Expression

GRP allows a group expression to be flagged by terminating it with a "flag" character (usually a minus sign although this can be changed, see section 5.7). If the last character in the group expression is a flag character, then the FLAG argument of routine GRP_GROUP is returned true. The flag character is stripped off the group expression before it is split up into elements, so the flag character itself does not get included in any of the names stored in the group.

A typical use of this facility might be to allow the user to request a further prompt for more names. For instance, in the example of section 1.2, the user may wish to specify more input file names than will fit on a single line. To allow this, the call to GRP_GROUP would be replaced with the following:

```
*  Loop round, prompting the user for group expressions
*  until one is found without a minus sign at the end, or an
*  error occurs.
      FLAG = .TRUE.
      DO WHILE( FLAG .AND. STATUS .EQ. SAI__OK )
         CALL GRP_GROUP( 'IN_FILES', GRP__NOID, IGRP1, SIZE1,
     :                   ADDED, FLAG, STATUS )

*  Cancel the parameter association to get a new group
*  expression on the next call to GRP_GROUP.
         CALL PAR_CANCL( 'IN_FILES', STATUS )

      END DO
```

The user could then request a further prompt by appending a minus sign to the end of the group expression, as follows:

```
NEW_FILE,A_*2|RAW|FLAT|,^LIST.DAT-
```

The names obtained at each prompt are appended to the end of the group, which expands as necessary.

Note, if the final element in a group expression is an indirection element, the flag character may be placed at the end of the last record in the indicated text file. For instance, instead of giving:

```
^LIST.DAT-
```

where the file LIST.DAT contains the single record

```
RED,GREEN,BLUE
```

a user could "hard-wire" a flag character on to the end of LIST.DAT so that it contains:

```
RED,GREEN,BLUE-
```

### 4.7 Comments Within Group Expressions

It is often useful to mix comments with names, particularly within a text file. All group expressions (whether obtained from the environment or from a text file or as an argument) are truncated if a "comment" character is found (usually '#" but this can be changed, see section 5.7). Anything occurring after such a character is ignored. In a text file, the comment is assumed to extend from the comment character to the end of the line, so a new group expression may be given on the next line. Note, blank lines are *not* ignored. Each blank line within a text file will result in a blank name being added to the group.

### 4.8 Escaping the Special Characters in a Group Expression

If is possible to specify that a given character be used as an "escape character" within group expressions. This facility is normally suppressed, but an application can choose to switch it on by assigning a value to the ESCAPE control character associated with a group (see section 5.7). If this is done, any special meaning associated with a character within a group expression is ignored if the character is preceeded by an escape character. The escape characters themselves are not included in the resulting names if they preceed any of the other "special" control characters. Note, escape characters which do not preceed another control character *are* included in the resulting names.

For instance, the group expression:

```
* | A
```

would normally result in an error because the "|" character would be taken as the start of an incomplete specification for some editing to apply to the preceeding text (assuming the application has not changed the default editing behaviour). If, in fact, the user wants this string to be accepted as a literal string (maybe representing a Unix piping operation for instance), then the "|" should be escaped. Assuming the application chooses to use the backslash character "\" as the escape character, then this can be done by entering the following group expression:

```
* \| A
```

The "\" character results in the "|" character being treated as part of the required string, rather than as the start of an editing specification. The string returned to the application is then "`* | A`" (note, the escape character has been removed). Any escape characters which do not preceed special characters are included literaly in the returned string. So, for instance, if the group expression was:

```
\* \| A
```

the string "`\* | A`" would be returned to the application.

All escape characters within a section of a group expression can be ignored by using the special strings "<!!" and "!!>" to mark the start and end of the section.

### 4.9 The Order of Names Within a Group

Names are stored within a group in the order in which they are specified in the group expression. For instance, if the file F1.DAT contained the following two records:

```
A,^F2
B,C
```

and the file F2.DAT contained the following three records:

```
D
E
F
```

then the group expression:

```
X,^F1.DAT,Y
```

would result in the names being added to the group in the following order

```
 X
 A
 D
 E
 F
 B
 C
 Y
```

The contents of the two indirection files have been inserted at the position at which the corresponding indirection element occurred. Names resulting from the expansion of modification elements are similarly inserted into the list at the position at which the modification element occurred. The modified names are stored in the same order as the names within the group upon which the modification was based. For example, if the above group is used as the basis for modification, then the group expression:

```
U,*_2,V
```

would give rise to the group:

```
U
X_2
A_2
D_2
E_2
F_2
B_2
C_2
Y_2
V
```

# 5 Using GRP Routines

Many of the points described below are illustrated by the example application in section 1.2. Further details can be found in the subroutine specifications in appendix C.

## 5.1 Symbolic Constants and Status Values

The GRP package has associated with it various symbolic constants defining such things as the required length of various character variables, an invalid GRP identifier value, etc. These values consist of a name of up to 5 characters prefixed by "GRP__" (note the *double* underscore), and can be made available to an application by including the line:

```
INCLUDE 'GRP_PAR'
```

Another set of symbolic constants is made available by the statement:

```
INCLUDE 'GRP_ERR'
```

These values have the same format of those contained in GRP_PAR, but define error conditions which can be generated within the GRP package. Applications can compare the STATUS argument with these values to check for specific error conditions. These values are described in appendix D.

## 5.2 Creating and Deleting Groups

New groups can be created by calling GRP_NEW which returns an identifier for the new group. The group is initially empty but names can be added immediately using any of the methods described in section 5.3. The calling application must provide a "type" for the group when calling GRP_NEW. This type is not used directly by the GRP system, but is provided as a means for applications to differentiate between different *types* of groups (for instance, groups holding file names and groups holding names of astronomical objects). No restrictions are placed on the strings which can be used for group types. The type of a group can be recalled at any time using routine GRP_GTYPE, and a group can be given a new type by calling GRP_PTYPE.

New groups are also created by the routines GRP_COPY, GRP_REMOV and GRP_PURGE, but they differ from GRP_NEW in that the created group is based on a previously existing group. GRP_COPY creates a new group containing a copy of a subsection of an old group, GRP_REMOV creates a new group containing a copy of the whole of an old group, but excluding all occurrences of a given name, and GRP_PURGE creates a new group containing a purged copy of the whole of an old group (i.e. there are no duplicate entries in the new group). The new groups inherit the type string, control characters and case sensitivity of the old groups, but do not inherit any other attributes (such as owner/slave relationships - see section 5.8).

The routine GRP_DELET deletes a group, thus freeing the GRP identifier and the internal resources used by the group. As well as deleting the group identified by the specified GRP identifier, it also deletes any groups *associated* with the specified group (by means of an owner-slave relationship as described in section 5.8).

To avoid running out of group identifiers, applications should always delete all groups which they have created (using GRP_DELET) before terminating, even if an error status exists.

### 5.3 Storing Names in a Group

There are several routines which can be used to store names in a group.

The routine GRP_PUT stores a set of names starting at a given index within a group. Any previous names with the same indices are over-written, and the group is extended as necessary. Note, an application must supply the *literal* names to be stored, as modification and indirection elements cannot be used with GRP_PUT. For instance, if the string "^LIST.DAT" was stored in a group using GRP_PUT, the ^character would *not* cause names to be read from the file LIST.DAT. Instead the string would be stored in the group exactly as supplied.

The routine GRP_PUT1 is like GRP_PUT except that it stores just a single name, and so has a simpler interface.

The routine GRP_GROUP obtains a group expression from the environment using a specified parameter and expands it into a list of literal names. These names are appended to the end of a group, which must previously have been created. An existing group may be specified as the basis group for any modification elements included in the group expression. If such a group is not supplied, all elements are stored literally as names. GRP_GROUP returns the total number of names in the group, together with the number of names which it has added. It also returns a logical argument indicating if the group expression was "flagged" (see section 4.6).

The routine GRP_GRPEX performs the same function as GRP_GROUP except that the group expression is supplied as an argument instead of being obtained through the parameter system. This routine can thus be used in stand-alone applications.

The following code fragment shows an example of the use of GRP_GRPEX and GRP_PUT. It is assumed that the text file FRIENDS.LIS exists and contains the single record "FRED,BERT".

```
      INTEGER IGRP, STATUS, SIZE, ADDED
      LOGICAL FLAG
      CHARACTER * ( GRP__SZNAM ) NAMES( 3 )

      ...

      CALL GRP_NEW( 'A test group', IGRP, STATUS )

      CALL GRP_GRPEX( '^FRIENDS.LIS', GRP__NOID, IGRP, SIZE, ADDED, FLAG,
     :                STATUS )

      CALL GRP_GRPEX( 'HAROLD', GRP__NOID, IGRP, SIZE, ADDED, FLAG,
     :                STATUS )
      ...

      NAMES( 1 ) = 'TOM'
      NAMES( 2 ) = 'DICK'
      NAMES( 3 ) = 'HARRY'

      ...

      CALL GRP_PUT( IGRP, 3, NAMES, 2, STATUS )
```

The call to GRP_NEW creates a new, empty group. The first call to GRP_GRPEX read the two names from the file FRIENDS.LIS and stores them with the following indices:

```
1 - FRED
2 - BERT
```

Both SIZE and ADDED are returned equal to two. The second call to GRP_GRPEX appends the name **HAROLD** to the group, so that the group becomes:

```
1 - FRED
2 - BERT
3 - HAROLD
```

SIZE is returned equal to 3 and ADDED equal to 1. The call to GRP_PUT then stores the name **TOM** at index 2 (over-writing **BERT**), and **DICK** at index 3 (over-writing **HAROLD**). The group is then extended and **HARRY** is stored at index 4 so that the group becomes:

```
1 - FRED
2 - TOM
3 - DICK
4 - HARRY
```

## 5.4 Retreiving Names from a Group

GRP_GET will retrieve a set of names from a group starting at a given index. An error is reported if a range of indices extending beyond the current size of the group is specified. If a character variable which receives a name has a declared length shorter than the stored name, then the name is truncated to the length of the character variable, but no error is reported.

A group can be searched for a specified name by calling GRP_INDEX, which returns the index of the name within the specified group. If the name does not exist within the group, the returned index is set to zero but no error is reported.

The routines GRP_LIST and GRP_LISTF each create a text file containing a subset of the names stored in a specified group. The names are written one per record, and a comment is stored as the first record. The difference between GRP_LIST and GRP_LISTF is that the former obtains the name of the text file to be created through the parameter system, whereas the latter requires the calling application to provide it. These routines are useful for passing groups of names between successive applications. For instance, the first application may create a text file holding a group of names using GRP_LIST, and the second application may then call GRP_GROUP, allowing the file created by the previous application to be specified within an indirection element.

A single name can also be retrieved from a group using routine GRP_INFOC, giving the value "NAME" for the argument ITEM (see below).

## 5.5 Retrieving Attributes of Names

Each name has associated with it various attributes, which can be retrieved using GRP_INFOC (for attributes which take character values) and GRP_INFOI (for attributes which take integer values). In both cases, the particular attribute required is specified by the argument ITEM. Attributes of a name are carried round with the name if it is copied using GRP_COPY, GRP_REMOV or GRP_PURGE. The following attributes are currently used:

**MODGRP** - An integer which has the value GRP__NOID unless the name was specified using a modification element. Otherwise, it is the identifier for the group which was used as the basis for the modification element (see section 4.4).

**MODIND** - An integer value which is zero unless the name was specified using a modification element. Otherwise, it is the index of the name within the basis group (i.e. the group identified by the MODGRP attribute) which was modified in order to generate the specified name.

**DEPTH** - An integer value which is zero unless the name was specified using an indirection element. Otherwise, it is the number of levels of indirection at which the literal name was specified. For instance, consider the case where a prompt issued by GRP_GROUP is responded to with the group expression

```
^FILE1.DAT
```

and the file FILE1.DAT contains the single record "NAME1,^FILE2.DAT", and the file FILE2.DAT contained the single record "NAME2". The DEPTH attribute would have a value of 1 for name **NAME1** indicating that the name was given within a file specified directly within the group expression, and a value of 2 for name **NAME2** indicating that the name was given within a file which was itself given within another file.

**FILE** - A character value which is returned blank unless the name was specified using an indirection element. Otherwise, it is the name of the file in which the name was literally stored. Thus, in the above example it would be FILE1.DAT for name **NAME1** and FILE2.DAT for **NAME2**.

**NAME** - A character value equal to the name itself.

## 5.6 Deleting Names

Names can be deleted from a group using GRP_REMOV, which deletes all occurrences of a specified name within a group, creating a new group to hold the results (the input group is left unchanged).

Names with a specified range of indices can be deleted from a group using GRP_COPY, as in the following example:

```
CALL GRP_COPY( IGRP, 3, 5, .TRUE., ITEMP, STATUS )
CALL GRP_DELET( IGRP, STATUS )
IGRP = ITEMP
```

The call to GRP_COPY creates a copy of the group identified by IGRP, excluding the names with indices between 3 and 5. The copy is stored in a new group, the identifier for which is returned in ITEMP. The names which used to have indices greater than 5 are shuffled down to fill up the gap (name 6 in the old group becomes name 3 in the new group, etc). The above example then deletes the original group, and copies the new group identifier into the variable previously used to store the identifier to the old group.

Names can also be deleted using GRP_SETSZ which reduces the size of a group. Thus, if a group with size of 6 is reduced to a size of 4, then the names with indices 5 and 6 are deleted.

Routine GRP_PURGE deletes duplicate names within a group, creating a new group to hold the results.

## 5.7 Changing the Characters Used to Control the Syntax of Group Expressions

A set of "control characters" are used to indicate various items of syntax within a group expression. Each group is created with a set of default control characters, but these can be changed by subsequent calls to GRP_SETCC. Each group has its own set of control characters which can be different to those of other groups, and they can be inspected using routine GRP_GETCC. When calling these routines, specific control characters are represented by the following names:

**INDIRECTION** - The character used to indicate that the string which follows is not a literal name but is the name of a text file from which further names are to be read. The INDIRECTION control character defaults to "^".

**COMMENT** - The character used to indicate that the remainder of the group expression (or record within a text file) is a comment and is to be ignored. The COMMENT control character defaults to "#".

**DELIMITER** - The character used to delimit elements within a group expression. The DELIMITER control character defaults to ",".

**NAME_TOKEN** - The character used within a modification element as a token for each name in the basis group. The NAME_TOKEN character defaults to "*".

**SEPARATOR** - The character used to separate old and new substitution strings when editing a group of names. The SEPARATOR character defaults to "|".

**OPEN_NEST** - The character used to open a nest within a group expression, within which any DELIMITER characters are ignored. The OPEN_NEST character defaults to "(".

**CLOSE_NEST** - The character used to close a nest within a group expression, within which any DELIMITER characters are ignored. The CLOSE_NEST character defaults to ")".

**FLAG** - The character which can be used to flag selected group expressions by appending it to the end of the group expression. The presence of the flag is communicated to the calling application by means of the FLAG argument of routines GRP_GROUP and GRP_GRPEX. The FLAG character defaults to "-".

**OPEN_KERNEL** - The character used to open a kernel within an element. The OPEN_KERNEL character defaults to "{".

**CLOSE_KERNEL** - The character used to close a kernel within an element. The CLOSE_KERNEL character defaults to "}".

**NULL** - This is a character which can be assigned to any other control character to suppress use of that control character (see below). The NULL character defaults to "%".

**ESCAPE** - This is a character which can be used to "escape" another control character within a group expression. If a GRP control character is encountered within a group expression, its special meaning is ignored and it is treated literally if it is preceeded by an escape character. Such escape characters are removed before storing the corresponding strings in the returned group (escape character which do not preceed another control character are not removed). Escape characters can themselves be escaped. By default, the escape character is equal to the null character (i.e. escaping is not available by default).

### 5.7.1 Suppressing the Use of Selected Control Characters

Applications may sometimes want to suppress the use of certain control characters. For instance, an application which stores lines of text within a group may want to suppress the division of each group expression into separate elements, so that the whole group expression is stored as a single name within the group. One way to do this would be to set the DELIMITER control character to a character which is guaranteed not to occur within the supplied text. However, such a character may not exist. A more flexible approach is to assign the value of the NULL control character to the DELIMITER control character. For instance, if the NULL and DELIMITER characters are both set to "%", then the group expression is not split up into elements delimited by "%" signs. Note, any "%" signs in the group expression will be included in the name stored in the group without any change. The NULL character can be assigned to any of the other control characters to suppress the use of those control character. The only restriction is that if one of the two control characters OPEN_NEST and CLOSE_NEST is assigned the NULL value, then the other one must also be assigned the NULL value. The same restriction applies to the OPEN_KERNEL and CLOSE_KERNEL control characters.

## 5.8 Creating Associations Between Groups

Each group may be "owned" by another group. This idea of ownership may be used to establish associations between groups. Many groups may be associated together into a sort of "super-group", or group of groups, by making one group the owner of another, which in turn is the owner of another, and so on. These associations are provided for the benefit of the calling application, which can impose any meaning it likes on the associations. The only significance of these associations within the GRP package is that when a group is deleted (using GRP_DELET), all other groups in the same "super-group" (i.e. all groups which have an association, either directly, or indirectly, with the specified group) are also deleted.

An example of the use of these associations is to provide a means of attaching supplemental information to a group describing the global properties of the group. For instance, consider an application which uses groups to store information about spiral galaxies. Group A may contain the names **1232**, **628**, **5364**, and group B may contain **81**, **31**. It may be necessary to store supplemental information describing each of these groups, for instance the catalogue to which the numbers in groups A and B refer, the morphological type, and so. A new group C could be created holding the names **NGC** and **ScI**, giving the catalogue and type of the galaxies in group A. This group could then be associated with group A. Similarly a group D could be created holding the names **Messier** and **Sb**, and associated with group B.

Four routines are included in the GRP package for handling these associations; GRP_SOWN establishes one group as owner of another, GRP_OWN returns the identifier of the group which owns a specified group, GRP_SLAVE returns the identifier of the group which is owned by (i.e. is a slave of) a specified group, and GRP_HEAD returns the group at the head of an owner-slave chain. Groups are "free" (i.e. are not owned by any other group) when they are created. A group which has had an owner established for it can be made free again by calling GRP_SOWN giving a null group (GRP__NOID) as the "owner". A group may not be owned by more than one group, and may itself own a maximum of one group. It is thus possible to establish "chains" of owner-slave associations (but not "tree structures"). All the groups in such a chain form a "super-group" of inter-dependant groups. When an attempt is made to delete any group within such a super-group, all the other groups in the same super-group are also deleted whether higher or lower in the chain.

The owner-slave relationship is strictly one way; a group cannot own its own owner, or equivalently be a slave of its own slave. If an attempt is made to set up such an association using GRP_SOWN, then an error is reported, and STATUS is returned set to GRP__SLAVE. Thus if group A is owned by group B, and group B is owned by group C, group C may not be owned by either group A or group B.

# 6 Compiling and Linking

To compile and link an ADAM application with the GRP package, the following commands should be used (see SUN/144):

```
% grp_dev
% alink adamprog.f ‘grp_link_adam‘
% grp_dev remove
```

Note the use of *opening* apostrophes (‘) rather than the more common closing apostrophe (’). To compile and link a stand-alone application with the GRP package, the following commands should be used:

```
% grp_dev
% f77 prog.f -o prog ‘grp_link‘
% grp_dev remove
```

This produces an executable image called **prog**. The `grp_dev` command creates soft links to the include files GRP_PAR and GRP_ERR within the current directory. These are removed by the `grp_dev remove` command.

The ADAM and stand-alone versions of the GRP_ system differ, in that those routines which use ADAM facilities (*i.e.* those listed in B.10) are not available in the stand-alone version.

# A Alphabetical List of Routines

**GRP_DELET**( IGRP, STATUS )
*Delete a group*

**GRP_COPY**( IGRP1, INDXLO, INDXHI, REJECT, IGRP2, STATUS )
*Create a new group containing a copy of a section of an old group*

**GRP_GET**( IGRP, INDEX, SIZE, NAMES, STATUS )
*Retrieve names from a group*

**GRP_GETCC**( IGRP, CCLIST, CC, STATUS )
*Return control characters for a given group*

**GRP_GETCS**( IGRP, SENSIT, STATUS )
*Determine if a group is case sensitive or not*

**GRP_GROUP**( PARAM, IGRP1, IGRP2, SIZE, ADDED, FLAG, STATUS )
*Add names obtained from the environment into a group*

**GRP_GRPEX**( GRPEXP, IGRP1, IGRP2, SIZE, ADDED, FLAG, STATUS )
*Add names specified by a given group expression into a group*

**GRP_GRPSZ**( IGRP, SIZE, STATUS )
*Return the current size of a group*

**GRP_GTYPE**( IGRP, TYPE, STATUS )
*Return the type string stored with a group*

**GRP_HEAD**( IGRP1, IGRP2, STATUS )
*Find the group at the head of an owner-slave chain*

**GRP_INDEX**( NAME, IGRP, START, INDEX, STATUS )
*Find the index of a given name within a group*

**GRP_INFOC**( IGRP, INDEX, ITEM, VALUE, STATUS )
*Return an item of character information relating to a specific name*

**GRP_INFOI**( IGRP, INDEX, ITEM, VALUE, STATUS )
*Return an item of integer information relating to a specific name*

**GRP_LIST**( PARAM, INDXLO, INDXHI, COMNT, IGRP, STATUS )
*Write all the names in a group out to a text file specified by the environment*

**GRP_LISTF**( FILENM, INDXLO, INDXHI, COMNT, IGRP, STATUS )
*Write all the names in a group out to a text file specified by the calling application*

**GRP_MSG**( TOKEN, IGRP, INDEX )
*Assign a group element to a message token*

**GRP_NEW**( TYPE, IGRP, STATUS )
*Create a new, empty group*

**GRP_OWN**( IGRP1, IGRP2, STATUS )
*Return the identifier of the group which "owns" a specified group*

**GRP_PTYPE**( IGRP, TYPE, STATUS )
*Store a new type string with a group*

**GRP_PURGE**( IGRP1, IGRP2, STATUS )
*Remove duplicate entries from a group, putting the results in a new group*

**GRP_PUT**( IGRP, SIZE, NAMES, INDEX, STATUS )
*Store a list of explicit names in a group*

**GRP_PUT1**( IGRP, NAME, INDEX, STATUS )
*Store a single name in a group*

**GRP_REMOV**( IGRP1, NAME, IGRP2, STATUS )
*Remove all occurrences of a given name from a group, putting the results in a new group*

**GRP_SETCC**( IGRP, CCLIST, CC, STATUS )
*Set up new control characters for a group*

**GRP_SETCS**( IGRP, SENSIT, STATUS )
*Establish a group as case sensitive or case insensitive*

**GRP_SETSZ**( IGRP, SIZE, STATUS )
*Reduce the size of a group*

**GRP_SLAVE**( IGRP1, IGRP2, STATUS )
*Return the identifier of the group which is owned by a specified group*

**GRP_SOWN**( IGRP1, IGRP2, STATUS )
*Establish one group as "owner" of another group*

**GRP_VALID**( IGRP, VALID, STATUS )
*Check if a group identifier is valid*

# B Classified List of Routines

## B.1 Creating and Deleting Groups

**GRP_DELET**( IGRP, STATUS )
*Delete a group*

**GRP_COPY**( IGRP1, INDXLO, INDXHI, REJECT, IGRP2, STATUS )
*Create a new group containing a copy of a section of an old group*

**GRP_NEW**( TYPE, IGRP, STATUS )
*Create a new, empty group*

**GRP_PURGE**( IGRP1, IGRP2, STATUS )
*Remove duplicate entries from a group, putting the results in a new group*

**GRP_REMOV**( IGRP1, NAME, IGRP2, STATUS )
*Remove all occurrences of a given name from a group, putting the results in a new group*

## B.2 Enquiring Group Attributes

**GRP_GETCC**( IGRP, CCLIST, CC, STATUS )
*Return control characters for a given group*

**GRP_GETCS**( IGRP, SENSIT, STATUS )
*Determine if a group is case sensitive or not*

**GRP_GRPSZ**( IGRP, SIZE, STATUS )
*Return the current size of a group*

**GRP_GTYPE**( IGRP, TYPE, STATUS )
*Return the type string stored with a group*

**GRP_HEAD**( IGRP1, IGRP2, STATUS )
*Find the group at the head of an owner-slave chain*

**GRP_OWN**( IGRP1, IGRP2, STATUS )
*Return the identifier of the group which "owns" a specified group*

**GRP_SLAVE**( IGRP1, IGRP2, STATUS )
*Return the identifier of the group which is owned by a specified group*

**GRP_VALID**( IGRP, VALID, STATUS )
*Check if a group identifier is valid*

## B.3 Setting Group Attributes

**GRP_DELET**( IGRP, STATUS )
*Delete a group*

**GRP_PTYPE**( IGRP, TYPE, STATUS )
*Store a new type string with a group*

**GRP_SETCC**( IGRP, CCLIST, CC, STATUS )
*Set up new control characters for a group*

**GRP_SETCS**( IGRP, SENSIT, STATUS )
*Establish a group as case sensitive or case insensitive*

**GRP_SETSZ**( IGRP, SIZE, STATUS )
*Reduce the size of a group*

**GRP_SOWN**( IGRP1, IGRP2, STATUS )
*Establish one group as "owner" of another group*

## B.4 Enquiring Name Attributes

**GRP_INDEX**( NAME, IGRP, START, INDEX, STATUS )
*Find the index of a given name within a group*

**GRP_INFOC**( IGRP, INDEX, ITEM, VALUE, STATUS )
*Return an item of character information relating to a specific name*

**GRP_INFOI**( IGRP, INDEX, ITEM, VALUE, STATUS )
*Return an item of integer information relating to a specific name*

## B.5 Retreiving Names

**GRP_GET**( IGRP, INDEX, SIZE, NAMES, STATUS )
*Retrieve names from a group*

**GRP_INFOC**( IGRP, INDEX, ITEM, VALUE, STATUS )
*Return an item of character information relating to a specific name*

**GRP_LIST**( PARAM, INDXLO, INDXHI, COMNT, IGRP, STATUS )
*Write all the names in a group out to a text file specified by the environment*

**GRP_LISTF**( FILENM, INDXLO, INDXHI, COMNT, IGRP, STATUS )
*Write all the names in a group out to a text file specified by the calling application*

**GRP_MSG**( TOKEN, IGRP, INDEX )
*Assign a group element to a message token*

## B.6 Storing Names

**GRP_COPY**( IGRP1, INDXLO, INDXHI, REJECT, IGRP2, STATUS )
*Create a new group containing a copy of a section of an old group*

**GRP_GROUP**( PARAM, IGRP1, IGRP2, SIZE, ADDED, FLAG, STATUS )
*Add names obtained from the environment into a group*

**GRP_GRPEX**( GRPEXP, IGRP1, IGRP2, SIZE, ADDED, FLAG, STATUS )
*Add names specified by a given group expression into a group*

**GRP_PUT**( IGRP, SIZE, NAMES, INDEX, STATUS )
*Store a list of explicit names in a group*

**GRP_PUT1**( IGRP, NAME, INDEX, STATUS )
*Store a single name in a group*

## B.7 Deleting Names

**GRP_COPY**( IGRP1, INDXLO, INDXHI, REJECT, IGRP2, STATUS )
*Create a new group containing a copy of a section of an old group*

**GRP_PURGE**( IGRP1, IGRP2, STATUS )
*Remove duplicate entries from a group, putting the results in a new group*

**GRP_REMOV**( IGRP1, NAME, IGRP2, STATUS )
*Remove all occurrences of a given name from a group, putting the results in a new group*

**GRP_SETSZ**( IGRP, SIZE, STATUS )
*Reduce the size of a group*

## B.8 Creation and Control of Identifiers

**GRP_DELET**( IGRP, STATUS )
*Delete a group*

**GRP_COPY**( IGRP1, INDXLO, INDXHI, REJECT, IGRP2, STATUS )
*Create a new group containing a copy of a section of an old group*

**GRP_NEW**( TYPE, IGRP, STATUS )
*Create a new, empty group*

**GRP_PURGE**( IGRP1, IGRP2, STATUS )
*Remove duplicate entries from a group, putting the results in a new group*

**GRP_REMOV**( IGRP1, NAME, IGRP2, STATUS )
*Remove all occurrences of a given name from a group, putting the results in a new group*

**GRP_VALID**( IGRP, VALID, STATUS )
*Check if a group identifier is valid*

## B.9 ADAM Parameter System Routines

**GRP_GROUP**( PARAM, IGRP1, IGRP2, SIZE, ADDED, FLAG, STATUS )
*Add names obtained from the environment into a group*

**GRP_LIST**( PARAM, INDXLO, INDXHI, COMNT, IGRP, STATUS )
*Write all the names in a group out to a text file specified by the environment*

## B.10 Debugging Routines

**GRP_ALARM**( IGRP, EVENT, STATUS )
*Notify the user of an event in the life of a watched group*

**GRP_WATCH**( IGRP, STATUS )
*Watch for events in the life of a specified group*

# C Routine Descriptions

# GRP_ALARM
## Notify the user of an event in the life of a watched group

**Description:**

This routine is called internally by the GRP library when a significant event happens in the life of a group that is being watched as a result of a call to GRP_WATCH. It is not intended to be called by the user. It's main purpose is to provide a break point target for debugging the creation and destruction of GRP groups.

**Invocation:**

```
CALL GRP_ALARM( IGRP, EVENT, STATUS )
```

**Arguments:**

**IGRP = INTEGER (Given)**
An identifier for the created group. GRP__NOID is returned if an error occurs.

**EVENT = CHARACTER * ( * ) (Given)**
This will be "CREATE" or "DELETE".

**STATUS = INTEGER (Given and Returned)**
The global status.

**Notes:**

- This routine attempts to run even if the global status value indicates an error has already occurred.

# GRP_COPY
## Copy a section of an existing group to a new group

**Description:**

A new group is created by copying a section of an existing group specified by the range of indices supplied in INDXLO and INDXHI. The whole group is used if INDXLO and INDXHI are both zero. The output group can be formed in one of two ways:

1) All names from the input group are copied to the output group except for those with indices in the given range.

2) Only those names from the input group which have indices within the given range are copied to the output group.

The method used is determined by the argument REJECT. Note, a name with a given index in the input group will have a different index in the output group if INDXLO is not 1 (or zero). The new group inherits the type, control characters and case sensitivity flag of the old group, but does not inherit any owner/slave relationships (see routine GRP_SOWN). If the input group is no longer required, it should be deleted using GRP_DELET.

**Invocation:**

```
CALL GRP_COPY( IGRP, INDXLO, INDXHI, REJECT, IGRP2, STATUS )
```

**Arguments:**

**IGRP1 = INTEGER (Given)**
A GRP identifier for the input group.

**INDXLO = INTEGER (Given)**
The lowest index to reject or to copy.

**INDXHI = INTEGER (Given)**
The highest index to reject or to copy.

**REJECT = LOGICAL ( Given)**
If reject is .TRUE., then names in the given range are rejected. Otherwise, names in the given range are copied.

**IGRP2 = INTEGER (Returned)**
A GRP identifier for the created group. Returned equal to GRP__NOID if an error occurs.

**STATUS = INTEGER (Given and Returned)**
The global status.

# GRP_DELET
## Delete a group from the GRP system

**Description:**

This routine releases the identifier and internal resources used by the specified group so that they can be used for another group. There is a limited number of groups available for use within an application, so each group should be deleted when it is no longer needed to avoid the possibility of reaching the limit.

Note, any parameter association used to establish the group is NOT cancelled.

This routine also deletes any groups which are related to the supplied group by means of "owner-slave" relationships established by calls to GRP_SOWN. All groups in the same "owner-slave" chain are deleted whether higher up or lower down than the supplied group.

This routine attempts to execute even if STATUS is bad on entry, although no further error report will be made if it subsequently fails under these circumstances.

**Invocation:**

```
CALL GRP_DELET( IGRP, STATUS )
```

**Arguments:**

**IGRP = INTEGER (Given and Returned)**
A GRP identifier for the group. Returned equal to GRP__NOID.

**STATUS = INTEGER (Given and Returned)**
The global status.

# GRP_GET
## Returns a set of names contained in a group

**Description:**

The names with indices between INDEX and INDEX+SIZE-1 (inclusive) contained in the given group are returned. An error is reported if the bounds of the group are exceeded, and STATUS is returned equal to GRP__OUTBN. If the group is case insensitive (as established by a call to GRP_SETCS) then the names are converted to upper case before being returned.

**Invocation:**

```
CALL GRP_GET( IGRP, INDEX, SIZE, NAMES, STATUS )
```

**Arguments:**

**IGRP = INTEGER (Given)**

A GRP identifier for the group.

**INDEX = INTEGER (Given)**

The lowest index for which the corresponding name is required.

**SIZE = INTEGER (Given)**

The number of names required.

**NAMES( SIZE ) = CHARACTER * ( * ) (Returned)**

The names held at the given positions in the group. The corresponding character variables should have declared length specified by the symbolic constant GRP__SZNAM. If the declared length is shorter than this, the returned names may be truncated, but no error is reported.

**STATUS = INTEGER (Given and Returned)**

The global status.

# GRP_GETCC
## Returns requested control characters for the specified group

**Description:**

Each group has associated with it several "control characters" which are the characters used to indicate various items of syntax within a group expression. These control characters can be changed at any time by calling GRP_SETCC. This routine returns the current values of a list of these control character. The individual characters are described in GRP__SETCC.

**Invocation:**

```
CALL GRP_GETCC( IGRP, CCLIST, CC, STATUS )
```

**Arguments:**

**IGRP = INTEGER (Given)**

A GRP identifier for the group for which the control characters are required.

**CCLIST = CHARACTER * ( * ) (Given)**

A comma separated list of control character names to be returned. See routine GRP_SETCC for a description of these names.

**CC = CHARACTER * ( * ) (Returned)**

A character variable to receive the requested list of control characters. The control characters are stored at adjacent indices within this character variable, starting at index 1. The characters are stored in the same order that they are specified in CCLIST. An error is reported if the character variable is not long enough to receive all the requested control characters.

**STATUS = INTEGER (Given and Returned)**

The global status.

# GRP_GETCS
## Determine the case sensitivity of a group

**Description:**

Checks whether a group is currently case sensitive, or case insensitive (see routine GRP_SETCS).

**Invocation:**

```
CALL GRP_GETCS( IGRP, SENSIT, STATUS )
```

**Arguments:**

**IGRP = INTEGER (Given)**
A GRP identifier for the group.

**SENSIT = LOGICAL (Returned)**
Returned .TRUE. if the group is case sensitive and .FALSE. otherwise.

**STATUS = INTEGER (Given and Returned)**
The global status.

# GRP_GROUP
## Append a list of names obtained from the environment to a previously created group

**Description:**

A group expression is obtained from the environment using the supplied parameter name. The expression is parsed to produce a list of names which are appended to the end of the group identified by IGRP2. Note, no permanent association between the parameter and the group exists. The parameter value can be cancelled without effecting the contents of the group. If an error occurs while parsing the group expression, the user is re-prompted for a new group expression.

If the group expression contains any modification elements, then the list of names added to the output group is based on the group identified by IGRP1. If IGRP1 is invalid (equal to the symbolic constant GRP__NOID for instance), then any elements with the syntax of a modification element are stored in the output group as a single literal name.

If the last character read from the group expression (or from a text file if the last element of the group expression is an indirection element) is equal to the current "flag" character for the group IGRP2 (see routine GRP_SETCC), then argument FLAG is returned set to .TRUE. Otherwise, it is returned set to .FALSE. The calling application can use this flag for any purpose (eg it may use it to indicate that the user wants to give more names). Note, the flag character itself is not included in the returned group.

**Invocation:**

```
CALL GRP_GROUP( PARAM, IGRP1, IGRP2, SIZE, ADDED, FLAG, STATUS )
```

**Arguments:**

**PARAM = CHARACTER * ( * ) (Given)**

The ADAM parameter with which to associate the group expression. This may be of any type.

**IGRP1 = INTEGER (Given)**

A GRP identifier for the group to be used as the basis for any modification elements which may be contained within the group expression obtained from the environment. This can be set to the symbolic constant GRP__NOID if modification elements are to be treated as literal names.

**IGRP2 = INTEGER (Given)**

A GRP identifier for the group to which the new names are to be appended.

**SIZE = INTEGER (Returned)**

The number of names in the returned group. It is returned equal to 1 if an error status exists on entry. If an error occurs during execution of this routine, then SIZE is returned equal to the size of the group on entry (unless the group has zero size on entry, in which case it is returned equal to 1).

**ADDED = INTEGER (Returned)**
The number of names added to the group as a result of the current call to this routine.

**FLAG = LOGICAL (Returned)**
.TRUE. if the last character in the group expression is equal to the current flag character for group IGRP2. Note, if this is the case, then the flag character itself is not included in the returned group. FLAG is returned .FALSE. if the last character is not a flag character. Returned .FALSE. if an error occurs.

**STATUS = INTEGER (Given and Returned)**
The global status.

**Notes:**

- A null value (!) can be given for the parameter to indicate that no more names are to be specified. The corresponding error is annulled before returning unless no names have been added to the group.

# GRP_GRPEX
## Append a list of names contained within a supplied group expression to a previously created group

**Description:**

The supplied group expression is expanded to produce a list of names which are appended to the end of the group identified by IGRP2.

If the group expression contains any modification elements, then the list of names added to the output group is based on the group identified by IGRP1. If IGRP1 is invalid (equal to the symbolic constant GRP__NOID for instance), then any elements with the syntax of a modification element are stored in the output group as a single literal name.

If the last character read from the group expression (or from a text file if the last element of the group expression is an indirection element) is equal to the current "flag" character for the group IGRP2 (see routine GRP_SETCC), then argument FLAG is returned set to true. Otherwise, it is returned set to false. The calling application can use this flag for any purpose (eg it may use it to indicate that the user wants to give more names). Note, the flag character itself is not included in the returned group.

**Invocation:**

```
CALL GRP_GRPEX( GRPEXP, IGRP1, IGRP2, SIZE, ADDED, FLAG, STATUS )
```

**Arguments:**

**GRPEXP = CHARACTER * ( * ) (Given)**

A group expression. This should not be longer than GRP__SZGEX. If it is, the surplus characters will be ignored.

**IGRP1 = INTEGER (Given)**

A GRP identifier for the group to be used as the basis for any modification elements which may be contained within the supplied group expression. This can be set to the symbolic constant GRP__NOID if modification elements are to be treated as literal names.

**IGRP2 = INTEGER (Given)**

A GRP identifier for the group to which the new names are to be appended.

**SIZE = INTEGER (Returned)**

The number of names in the returned group. It is returned equal to 1 if an error status exists on entry. If an error occurs during execution of this routine, then SIZE is returned equal to the size of the group on entry (unless the group has zero size on entry, in which case it is returned equal to 1).

**ADDED = INTEGER (Returned)**

The number of names added to the group by this routine.

**FLAG = LOGICAL (Returned)**

.TRUE. if the last character in the group expression is equal to the current flag character

for group IGRP2. Note, if this is the case, then the flag character itself is not included in the returned group. FLAG is returned .FALSE. if the last character is not a flag character. Returned .FALSE. if an error occurs.

**STATUS = INTEGER (Given and Returned)**
The global status.

# GRP_GRPSZ
## Returns the number of names in a group

**Description:**

This routine returns the number of names in a group.

**Invocation:**

```
CALL GRP_GRPSZ( IGRP, SIZE, STATUS )
```

**Arguments:**

**IGRP = INTEGER (Given)**

A GRP identifier for the group.

**SIZE = INTEGER (Returned)**

The number of names in the group. Returned equal to one if an error occurs.

**STATUS = INTEGER (Given and Returned)**

The global status.

# GRP_GTYPE
## Retrieve the type string stored with a group

**Description:**

The type string specified when the group was created is retrieved.

**Invocation:**

```
CALL GRP_GTYPE( IGRP, TYPE, STATUS )
```

**Arguments:**

**IGRP = INTEGER (Given)**

A GRP group identifier.

**TYPE = CHARACTER * ( * ) (Returned)**

The group type. The maximum allowable type length is given by the symbolic constant GRP__SZTYP. If the supplied variable is too short, the type is truncated but no error is reported. If an error occurs, the string is returned blank.

**STATUS = INTEGER (Given and Returned)**

The global status.

# GRP_HEAD
## Finds the group which is at the head of an owner-slave chain

**Description:**

This routine climbs the chain of owners starting at the group identified by IGRP1, until a group is found which has no owner. The identifier issued for this group is returned in IGRP2. If the group identified by IGRP1 has no owner, then IGRP2 is returned equal to IGRP1.

**Invocation:**

```
CALL GRP_HEAD( IGRP1, IGRP2, STATUS )
```

**Arguments:**

**IGRP1 = INTEGER (Given)**

A group identifier.

**IGRP2 = INTEGER (Returned)**

The identifier for the group which is at the head of the owner-slave chain. Returned equal to GRP__NOID if an error occurs.

**STATUS = INTEGER (Given and Returned)**

The global status.

# GRP_INDEX
## Searches for a given name and if found, returns its index

**Description:**

The group is searched for the given name, starting at the name with index given by START, and continuing to the end of the group. If it is found then the corresponding index within the group is returned. If it is not found, the index is set to zero, but no error status is generated. The search is case sensitive unless GRP_SETCS has been called to indicate that the group is case insensitive. If the section of the group searched contains the name more than once then the lowest index is returned.

**Invocation:**

```
CALL GRP_INDEX( NAME, IGRP, START, INDEX, STATUS )
```

**Arguments:**

**NAME = CHARACTER * ( * ) (Given)**
The name to be searched for.

**IGRP = INTEGER (Given)**
A GRP identifier for the group to be searched.

**START = INTEGER (Given)**
The lowest index to be checked.

**INDEX = INTEGER (Returned)**
The index of the name within the group. This number is greater than or equal to START if the name is found, and zero if it is not found.

**STATUS = INTEGER (Given and Returned)**
The global status.

# GRP_INFOC
## Retrieve an item of character information about a name

**Description:**

This routine returns an item of character information about a single name from a group. The item can be any one of those described under argument ITEM.

**Invocation:**

```
CALL GRP_INFOC( IGRP, INDEX, ITEM, VALUE, STATUS )
```

**Arguments:**

**IGRP = INTEGER (Given)**

A GRP group identifier.

**INDEX = INTEGER (Given)**

An index within the group specified by IGRP. If the supplied value is outside the bounds of the group, then a blank value is returned for VALUE, and an error is reported.

**ITEM = CHARACTER * ( * ) (Given)**

The name of an item of information. This can be any of the following (abbreviations are not allowed):

NAME - The name itself. If the group is case insensitive (as established by a call to routine GRP_SETCS) then the name is returned in upper case.

FILE - The text file within which the name was explicitly given. If the name was not specified within a file then FILE is returned blank.

**VALUE = CHARACTER * ( * ) (Returned)**

The requested item of information. If the supplied character variable is too short, the string is truncated. If an error occurs a blank value is returned.

**STATUS = INTEGER (Given and Returned)**

The global status.

# GRP_INFOI
## Retrieve an item of integer information about a name

**Description:**

This routine returns an item of integer information about a single name from a group. The item can be any one of those described under argument ITEM.

It can also return the current number of active GRP identifiers.

**Invocation:**

```
CALL GRP_INFOI( IGRP, INDEX, ITEM, VALUE, STATUS )
```

**Arguments:**

**IGRP = INTEGER (Given)**

A GRP group identifier.

**INDEX = INTEGER (Given)**

An index within the group specified by IGRP. If the supplied value is outside the bounds of the group, then a "null" value is returned for VALUE as described below, and an error is reported.

**ITEM = CHARACTER * ( * ) (Given)**

The name of an item of information. This can be any of the following (abbreviations are not allowed):

MODGRP - If the name was specified by means of a modification element, then the the GRP identifier of the group used as a basis for the modification element is returned in VALUE. If the name was not specified by a modification element, then GRP__NOID is returned. If INDEX is outside the bounds of the group, then a value of GRP__NOID is returned.

MODIND - If the name was specified by means of a modification element, then the index of the original name (upon which the returned name was based) is returned in VALUE. This is an index into the group identified by MODGRP. If MODGRP is returned equal to GRP__NOID, then MODIND will be zero.

DEPTH - The number of levels of indirection at which the name was specified is returned in VALUE. Names given explicitly within a group expression have a DEPTH value of zero. Names given explicitly within a DEPTH zero indirection element have DEPTH 1. Names given explicitly within a DEPTH 1 indirection element, have DEPTH 2, etc. If INDEX is out of bounds, then zero is returned.

NGRP - The number of GRP group identifiers currently in use. The values of the IGRP and INDEX arguments are ignored.

ACTIVE - Like NGRP, except that in addition to returning the number of GRP group identifiers currently in use, the numerical identifier values are also listed on standard output (using MSG_OUT).

ALLSIMPLE - A value of 1 is returned if all elements within the group were specified explicitly (i.e. by a literal name rather than by indirection or modification). Otherwise a value of zero is returned. The value of the INDEX argument is ignored.

**VALUE = INTEGER (Returned)**
The requested item of information.

**STATUS = INTEGER (Given and Returned)**
The global status.

# GRP_LIST
## Write names to a text file specified by the environment

**Description:**

A text file is created with a name obtained from the environment using the supplied parameter. The supplied comment is written to the file as the first record (so long as it is not blank), using the groups current comment character (see routine GRP_SETCC) . All the names stored within the specified group section are then written to the file, one name per record. If the group is case insensitive (as set up by a call to routine GRP_SETCS) then the names are written out in upper case, otherwise they are written out as supplied.

**Invocation:**

```
CALL GRP_LIST( PARAM, INDXLO, INDXHI, COMNT, IGRP, STATUS )
```

**Arguments:**

**PARAM = CHARACTER * ( * ) (Given)**

The parameter to be used to get the name of the text file to be created.

**INDXLO = INTEGER (Given)**

The low index limit of the group section. If both INDXLO and INDXHI are zero, then the entire group is used.

**INDXHI = INTEGER (Given)**

The high index limit of the group section.

**COMNT = CHARACTER * ( * ) (Given)**

A comment line to form the first record in the file. The text is prefixed with the group's current comment character before being written to the file.

**IGRP = INTEGER (Given)**

The GRP identifier for the group to be listed.

**STATUS = INTEGER (Given and Returned)**

The global status.

# GRP_LISTF
## Write names to a specified text file

**Description:**

A text file is created with the specified name. The supplied comment is written to the file as the first record (so long as it is not blank), using the groups current comment character (see routine GRP_SETCC). All the names stored within the specified group section are then written to the file, one name per record. If the group is case insensitive (as set up by a call to routine GRP_SETCS) then the names are written out in upper case, otherwise they are written out as supplied.

The routine GRP_LIST can be used if the file name is to be obtained through the parameter system.

**Invocation:**

```
CALL GRP_LISTF( FILENM, INDXLO, INDXHI, COMNT, IGRP, STATUS )
```

**Arguments:**

**FILENM = CHARACTER * ( * ) (Given)**
The name of the text file to be created.

**INDXLO = INTEGER (Given)**
The low index limit of the group section. If both INDXLO and INDXHI are zero, then the entire group is used.

**INDXHI = INTEGER (Given)**
The high index limit of the group section.

**COMNT = CHARACTER * ( * ) (Given)**
A comment line to form the first record in the file. The text is prefixed with a the groups current comment character before being written to the file.

**IGRP = INTEGER (Given)**
The GRP identifier for the group to be listed.

**STATUS = INTEGER (Given and Returned)**
The global status.

# GRP_MSG
## Assign an element of a group to a message token

**Description:**

The routine assigns a specified element of a GRP group to a message token for use in constructing messages with the ERR_ and MSG_ routines (see SUN/104).

**Invocation:**

```
CALL GRP_MSG( TOKEN, IGRP, INDEX )
```

**Arguments:**

**TOKEN = CHARACTER * ( * ) (Given)**
Name of the message token.

**IGRP = INTEGER (Given)**
A GRP identifier for the group.

**INDEX = INTEGER (Given)**
The index of the element to assign to the message token.

**Notes:**

- This routine has no inherited status argument. It will always attempt to execute, and no error will be reported if it should fail (although the message token will be assigned a blank string).

# GRP_NEW
## Create a new empty group

**Description:**

A new empty group is created and an identifier to it is returned in IGRP. The string supplied in TYPE is stored with the group, and should be used to store a description of the contents of the group (see also routines GRP_GTYPE and GRP_PTYPE).

The created group has the default control characters described in routine GRP_SETCC, and has no "owner" group (see GRP_OWN).

**Invocation:**

```
CALL GRP_NEW( TYPE, IGRP, STATUS )
```

**Arguments:**

**TYPE = CHARACTER * ( * ) (Given)**

A descriptive string to be associated with the group. The maximum length for a TYPE string is given by GRP__SZTYP. Supplied characters beyond this length are ignored.

**IGRP = INTEGER (Returned)**

An identifier for the created group. GRP__NOID is returned if an error occurs.

**STATUS = INTEGER (Given and Returned)**

The global status.

# GRP_OWN
## Returns the identifier of the group which owns the specified group

**Description:**

If the group identified by IGRP1 has had an "owner" group established for it by a call to GRP_SOWN, then the identifier of the owner group is returned in IGRP2. Otherwise, the value GRP__NOID is returned (but no error is reported).

**Invocation:**

```
CALL GRP_OWN( IGRP1, IGRP2, STATUS )
```

**Arguments:**

**IGRP1 = INTEGER (Given)**

An identifier for the slave group whose owner is to be returned. If GRP__NOID is supplied, then IGRP2 is returned holding GRP__NOID and no error is reported.

**IGRP2 = INTEGER (Returned)**

An identifier for the group which owns the group identified by IGRP1. Returned equal to GRP__NOID if an error occurs.

**STATUS = INTEGER (Given and Returned)**

The global status.

# GRP_PTYPE
## Associate a new type string with a group

**Description:**

The given type string is stored with the group, replacing the previous value. The maximum length of the type string is given by symbolic constant GRP__SZTYP. If the supplied type string is longer than this, the title is truncated.

**Invocation:**

```
CALL GRP_PTYPE( IGRP, TYPE, STATUS )
```

**Arguments:**

**IGRP = INTEGER (Given)**
A GRP identifier for the group.

**TYPE = CHARACTER * ( * ) (Given)**
The group type.

**STATUS = INTEGER (Given and Returned)**
The global status.

# GRP_PURGE
## Purge duplicate entries from a group

**Description:**

This routine creates a new group based on a given existing group. The contents of the existing group are copied to the new group, but any duplicated names are only included once. The check for duplication is case sensitive unless the group has been declared case insensitive by a call to GRP_SETCS. The new group inherits the type, control characters case sensitivity flag of the old group, but does not inherit any owner/slave relationships (see routine GRP_SOWN).

Note, indices determined from the old group will in general not point to the same name in the new group. The old group should be deleted using GRP_DELET if it is no longer required.

**Invocation:**

```
CALL GRP_PURGE( IGRP1, IGRP2, STATUS )
```

**Arguments:**

**IGRP1 = INTEGER (Given)**

The GRP identifier for an existing group.

**IGRP2 = INTEGER (Returned)**

A GRP identifier for the created group. This group is a purged form of the group identified by IGRP1. A value of GRP__NOID is returned if an error occurs.

**STATUS = INTEGER (Given and Returned)**

The global status.

# GRP_PUT
## Put a given set of literal names into a group

**Description:**

The given names are stored in the group in the form in which they are supplied (including any control characters). They overwrite any previous names stored at the specified indices. The group is extended if the range of indices extends beyond the current size of the group. The names can be appended to the end of the group by giving INDEX a value of zero or one greater than the current size of the group. An error is reported if the names are added beyond the end of the group (i.e. if adding the names would result in a gap within the group for which no names would be defined).

**Invocation:**

```
CALL GRP_PUT( IGRP, SIZE, NAMES, INDEX, STATUS )
```

**Arguments:**

**IGRP = INTEGER (Given)**
A GRP identifier for the group.

**SIZE = INTEGER (Given)**
The size of the NAMES array.

**NAMES( SIZE ) = CHARACTER * ( * ) (Given)**
The names to be stored in the group. The first name is stored at the index given by INDEX, the last is stored at index INDEX+SIZE-1.

**INDEX = INTEGER (Given)**
The index at which to store the first name. A value of zero causes the names to be appended to the end of the group.

**STATUS = INTEGER (Given and Returned)**
The global status.

# GRP_PUT1
## Put a single name into a group

**Description:**

The given name is stored in the group in the form in which it is supplied (including any control characters). It overwrites any previous name stored at the specified index. The name can be appended to the end of the group by giving INDEX a value of zero or one greater than the current size of the group. An error is reported if the name is added beyond the end of the group (i.e. if adding the name would result in a gap within the group for which no names would be defined).

**Invocation:**

```
CALL GRP_PUT1( IGRP, NAME, INDEX, STATUS )
```

**Arguments:**

**IGRP = INTEGER (Given)**
A GRP identifier for the group.

**NAME = CHARACTER * ( * ) (Given)**
The name to be stored in the group.

**INDEX = INTEGER (Given)**
The index at which to store the name. A value of zero causes the names to be appended to the end of the group.

**STATUS = INTEGER (Given and Returned)**
The global status.

# GRP_REMOV
## Remove all occurrences of a given name from a group

**Description:**

A new group is created by copying the contents of an existing group, excluding any occurrences of a specified name. Note, a name with a given index in the input group will in general have a different index in the output group. The new group inherits the type, control characters and case sensitivity flag of the old group, but does not inherit any owner/slave relationships (see routine GRP_SOWN). If the input group is no longer required, it should be deleted using GRP_DELET.

**Invocation:**

```
CALL GRP_REMOV( IGRP1, NAME, IGRP2, STATUS )
```

**Arguments:**

**IGRP1 = INTEGER (Given)**

A GRP identifier for the input group.

**NAME = CHARACTER * ( * )(Given)**

The name to be removed. Leading blanks are significant, and case is also significant unless the group has been marked as case insensitive by calling GRP_SETCS.

**IGRP2 = INTEGER (Returned)**

A GRP identifier for the created group. Returned equal to GRP__NOID if an error occurs.

**STATUS = INTEGER (Given and Returned)**

The global status.

# GRP_SAME
## Determine if two GRP identifiers refer to the same group

**Description:**

The routines returns a flag indicating if the two supplied GRP identifiers refer to the same group.

**Invocation:**

```
CALL GRP_SAME( IGRP1, IGRP2, SAME, STATUS )
```

**Arguments:**

**IGRP1 = INTEGER (Given)**
The first GRP identifier.

**IGRP2 = INTEGER (Given)**
The second GRP identifier.

**SAME = LOGICAL (Returned)**
Returned set to .TRUE. if the two identifiers refer to the same group, and .FALSE. otherwise.

**STATUS = INTEGER (Given and Returned)**
The global status.

# GRP_SETCC
## Sets requested control characters for the specified group

**Description:**

Each group has associated with it several "control characters" which are the characters used to indicate various items of syntax within a group expression. These characters are listed and described in the "Notes" section below. The control characters are given default values when the group is created, but can be changed at any time by calling this routine.

Checks for particular control characters may be suppressed by assigning the NULL character to them. The NULL character is itself a control character which may be assigned a value using this routine. Some control characters form pairs, and an error is reported if only one member of a pair is assigned the NULL value. These pairs are OPEN_NEST and CLOSE_NEST, and OPEN_KERNEL and CLOSE_KERNEL.

If a blank value for argument CCLIST is supplied, then the default control characters described in the "Notes" section are re-established.

An error is reported if any two control characters are the same. The exception to this is that any number of control characters may have the same value as the NULL control character. If any error occurs, the control characters are left unaltered.

**Invocation:**

```
CALL GRP_SETCC( IGRP, CCLIST, CC, STATUS )
```

**Arguments:**

**IGRP = INTEGER (Given)**

A GRP identifier for the group to which the control characters refer.

**CCLIST = CHARACTER * ( * ) (Given)**

A comma separated list of names specifying which control character are to be altered. Unambiguous abbreviations may be used. A blank value causes all control characters to be reset to the default values.

**CC = CHARACTER * ( * ) (Given)**

A list of the new control characters, in the same order as the corresponding names in CCLIST. Note, if CCLIST contains N names, then the first N characters are used out of the string specified by CC. If the total length (including any trailing blanks) of CC is less than N, then an error is reported.

**STATUS = INTEGER (Given and Returned)**

The global status.

**Notes:**

The following names are used to refer to the individual control characters:

- INDIRECTION: (Default "^") The name given to the character used to indicate that an element of a group expression is the name of a text file from which further elements should be read.

- COMMENT: (Default "#") The name given to the character used to introduce comments in group expressions.
- DELIMITER: (Default ",") The name given to the character used to delimit elements within group expressions. Note, delimiters within group expressions are ignored if they occur within matched nesting characters (see OPEN_NEST and CLOSE_NEST below).
- NAME_TOKEN: (Default "*") The name given to the character used as a token for input names in a modification element.
- SEPARATOR: (Default "|") The name given to the character used to separate the substitution strings within a modification element.
- OPEN_NEST: (Default "(") The name given to the character used to open a "nest" within a group expression. Any delimiter characters occurring within matched nesting characters are ignored.
- CLOSE_NEST: (Default ")") The name given to the character used to close a "nest" within a group expression.
- FLAG: (Default "-") The name given to a character which can be appended to the end of a group expression in order to "flag" that expression. The interpretation of this flag is left up to the application.
- OPEN_KERNEL: (Default "{") The name given to the character used to open a "kernel" within a group expression.
- CLOSE_KERNEL: (Default "}") The name given to the character used to close a "kernel" within a group expression.
- NULL: (Default "%") The name given to the character which can be assigned to other control characters to suppress checks for those control characters. If this is changed, any other characters currently set to the null character are also changed to the new NULL character.
- ESCAPE: (Default to the NULL character) The name given to the character which can be used to escape control characters within a group expression.

# GRP_SETCS
## Establish the case sensitivity of a group

**Description:**

When a group is created using GRP_NEW, it is initially case sensitive. This routine can be called to make it case insensitive, or to make it case sensitive again.

All names stored within a case sensitive group are used in the same form as they were supplied by the environment or application. If the group is case insensitive the upper case equivalent is used when any reference is made to a name stored within the group, and all comparisons between strings (such as performed within routines GRP_INDEX and GRP_PURGE for instance) related to that group are performed without reference to case.

Note, names are always stored in the form they are given. Any case conversion takes place when the names are read out of the group, not when they are put into the group. This means that groups can be changed at any time from being case insensitive to being case sensitive (or vice-versa).

**Invocation:**

```
CALL GRP_SETCS( IGRP, SENSIT, STATUS )
```

**Arguments:**

**IGRP = INTEGER (Given)**
A GRP identifier for the group.

**SENSIT = LOGICAL (Given)**
If .TRUE. then the group is made case sensitive. If .FALSE., then the group is made case insensitive.

**STATUS = INTEGER (Given and Returned)**
The global status.

# GRP_SETSZ
## Reduce the size of a group

**Description:**

This routine sets the size of the given group to the specified value. The new size must be less than or equal to the old size. The names with indices greater than the new size are lost. Other names remain unaltered.

**Invocation:**

```
CALL GRP_SETSZ( IGRP, SIZE, STATUS )
```

**Arguments:**

**IGRP = INTEGER (Given)**

A GRP identifier for the group.

**SIZE = INTEGER (Given)**

The new group size. If a negative value is given, then zero is used.

**STATUS = INTEGER (Given and Returned)**

The global status.

# GRP_SHOW
## List contents ofa group to the screen

**Description:**

The contents of the supplied group are listed to the screen, with one name on each line. If the group has a non-blank type string, it is dislayed first, prefixed with the groups current comment character (see routine GRP_SETCC). If SLAVES is .TRUE., then any slave group owned by the supplied group is also displayed. This is recursive - any slave group owned by a displayed slave group is also displayed.

**Invocation:**

```
CALL GRP_SHOW( IGRP, SLAVES, STATUS )
```

**Arguments:**

**IGRP = INTEGER (Given)**

The GRP identifier for the group to be displayed.

**SLAVES = LOGICAL (Given)**

If TRUE., then the entire chain of slaved groups owned by the specified group are also displayed.

**STATUS = INTEGER (Given and Returned)**

The global status.

**Notes:**

- If the group is case insensitive (as set up by a call to routine GRP_SETCS) then the names are written out in upper case, otherwise they are written out as supplied.

# GRP_SLAVE
## Returns the identifier of the group which is owned by the specified group

**Description:**

If the group identified by IGRP1 has had a "slave" group established for it by a call to GRP_SOWN, then the identifier of the slave group is returned in IGRP2. Otherwise, the value GRP__NOID is returned (but no error is reported).

**Invocation:**

```
CALL GRP_SLAVE( IGRP1, IGRP2, STATUS )
```

**Arguments:**

**IGRP1 = INTEGER (Given)**

An identifier for the owner group whose slave is to be returned.

**IGRP2 = INTEGER (Returned)**

An identifier for the group which is owned by the group identified by IGRP1. Returned equal to GRP__NOID if an error occurs.

**STATUS = INTEGER (Given and Returned)**

The global status.

# GRP_SOWN
## Establish one group as the owner of another group

**Description:**

This routine establishes one specified group as "owner" of another specified group. An error is reported if the group is already owned by another group. An error is also reported if the "owner" group already owns a slave.

This routine may also be used to cancel an "owner-slave" relationship, by specifying IGRP2 as GRP__NOID. The group identified by IGRP1 then becomes a "free" group (i.e. has no owner). An error is reported if IGRP1 identifies a group which is already free.

**Invocation:**

```
CALL GRP_SOWN( IGRP1, IGRP2, STATUS )
```

**Arguments:**

**IGRP1 = INTEGER (Given)**
The identifier of the "slave" group which is to have an owner established for it. An error is reported if an invalid identifier is given.

**IGRP2 = INTEGER (Given)**
The identifier of the group which is to be established as the owner of the group identified by IGRP1.

**STATUS = INTEGER (Given and Returned)**
The global status.

**Notes:**

- There is a restriction on the use of owner-slave relationships, namely that a slave cannot own its own owner, either directly or indirectly. That is, if group A is owned by group B, and group B is owned by group C, then group C cannot be owned by either group B, or group A. An error is reported, if an attempt is made to set up such a relationship.
- When a group is deleted using GRP_DELET, all other groups in the same owner/slave chain, whether higher up or lower down, are also deleted. If a group is to be deleted without deleting all other related groups, then the group must be established as a "free" group (i.e. no owner) by calling this routine with IGRP2 set to GRP__NOID, before calling GRP_DELET.

# GRP_VALID
## Determine if a group identifier is valid

**Description:**

Argument VALID is returned .TRUE. if the group identified by IGRP is valid, and is returned .FALSE. otherwise.

**Invocation:**

```
CALL GRP_VALID( IGRP, VALID, STATUS )
```

**Arguments:**

**IGRP = INTEGER (Given)**
A GRP identifier.

**VALID = LOGICAL (Returned)**
The status of the group.

**STATUS = INTEGER (Given and Returned)**
The global status.

# GRP_WATCH
## Watch for events in the life of a specified group

**Description:**

This routine causes messages to be reported when a group with the given integer identifier is created or destroyed. It will also list the integer identifiers for all currently active groups.

Each event in the life of the group is repoirted by routine GRP_ALARM, and so a debugger break point can be set there to investigate such events.

**Invocation:**

```
CALL GRP_WATCH( IGRP, STATUS )
```

**Arguments:**

**IGRP= INTEGER (Given)**

The identifier for the group to be watched. If this is zero, the identifiers of all currently active groups are listed on standard output. NOTE, in the C interface this should be an integer, not a pointer to a Grp structure.

**STATUS = INTEGER (Given and Returned)**

The global status.

**Notes:**

- This routine attempts to run even if the global status value indicates an error has already occurred.

# D GRP Error Status Values

This appendix lists the STATUS values generated within the GRP package. Note, STATUS values generated by the packages listed in appendix E may also be returned.

**GRP__BADCC** - An ambiguous or un-recognised control character name has been supplied.

**GRP__BADIT** - An un-recognised item of information has been requested.

**GRP__BADME** - A modification element has been given which has an illegal format.

**GRP__DEEP** - The maximum depth of indirection has been exceeded.

**GRP__EMPTY** - The specified group is empty.

**GRP__FIOER** - A Fortran I/O error has occurred.

**GRP__FREE** - The group to be made free is already free.

**GRP__INTER** - An internal GRP error has occurred.

**GRP__INVID** - An invalid group identifier was given.

**GRP__NOLUN** - No free logical unit numbers available.

**GRP__NOMOR** - No more groups can be created.

**GRP__NULNM** - A zero length name was given.

**GRP__OUTBN** - A group has been indexed outside its bounds.

**GRP__OWNED** - The group to be enslaved is already owned.

**GRP__OWNER** - An attempt has been made to enslave a group to a group which already owns another group.

**GRP__SHORT** - A character variable supplied as an argument was too short.

**GRP__SLAVE** - An attempt has been made to make a group a slave of its own slave.

**GRP__SZINC** - An attempt has been made to increase the size of a group using GRP_SETSZ.

# E Packages Called From Within GRP

The GRP package makes use of subroutines from the PSX library (see SUN/121), the CHR library (see SUN/40) and the ERR library (see SUN/104).

# F Acknowledgements

Peter Draper is thanked for his work on porting the original IRH_ package to UNIX, and Rodney Warren-Smith is thanked for his extensive suggestions for improvements to the GRP_ package.

# G Changes and New Features in V3.7

The following changes have occurred in the GRP_ system since version 3.6:

(1) Addition of routine GRP_SAME.

(2) Addition of routine GRP_SHOW.

(3) Addition of routine GRP_ALARM.

(4) Addition of routine GRP_WATCH.

# H Changes and New Features in V3.6

The following changes have occurred in the GRP_ system since version 3.5:

(1) Addition of routine GRP_SLAVE.

# I Changes and New Features in V3.5

The following changes have occurred in the GRP_ system since version 3.4:

(1) Addition of routine GRP_MSG.

# J Changes and New Features in V3.4

The following changes have occurred in the GRP_ system since version 3.3:

(1) The GRP_INFOI routine has a new option to return a flag indicating if all elements in a group were supplied literally (i.e. not by indirection or modification).

# K Changes and New Features in V3.3

The following changes have occurred in the GRP_ system since version 3.2:

(1) The GRP_PUT1 routine has been added.

# L Changes and New Features in V3.2

The following changes have occurred in the GRP_ system since version 3.1:

(1) The GRP_INFOI routine can now return the total number of active GRP identifiers.

(2) Shell meta-characters (references to environment variables, for instance) may now be included within file names which specify indirection elements.

# M Changes and New Features in V3.1

The following changes have occurred in the GRP_ system since version 3.0:

(1) Sections of a group expression can now be marked as "verbatim text" by beginning the section with the string "<!!" and ending it with "!!>". All control characters within such a section are treated as literal characters (the delimiting strings themselves are thrown away).

# N Changes and New Features in V3.0

The following changes have occurred in the GRP_ system since version 2.0:

(1) The maximum number of groups which can be active at any one time has been increased from 100 to 500.

(2) An escape control character can now be defined which allows other control characters within a group expression to be treated literally (i.e. their usual special meaning is suppressed and they are trated like a normal text character if they are preceeded by an escape character). By default, no escape character is defined.

(3) If the NULL control character is changed using GRP_SETCC, then any other control characters which are equal to the old NULL character are changed to the new NULL character, so long as the same call to GRP_SETCC does not explicitly set their value to something else.

# O Changes and New Features in V2.0

The following changes have occurred in the GRP_ system since version 1.1:

(1) The facility for editing names which was previously restricted to use within modification elements, has been made available for use with any form of element. See section 4.

(2) Null names (i.e. names with zero length) are now accepted and treated as blank names.

# P Changes and New Features in V1.1

The following changes have occurred in the GRP_ system since version 1.0:

(1) A new routine GRP_HEAD has been introduced to simplify the task of finding the head of an owner-slave chain.

(2) Routines GRP_GROUP and GRP_GRPEX

- Argument FLAG is now returned false if an error occurs.
- If an error occurs, then argument SIZE is returned equal to the supplied size of the group identified by argument IGRP2. If the group has zero size on entry then argument SIZE is returned equal to one.