UNIVERSITY OF EDINBURGH
Edinburgh Regional Computing Centre

User Note 16

(January 1984)

Title:

Profile Scheme on EMAS 2900

Author: John M. Murison

Contact: Advisory Service

Software Support Category: B

Synopsis

The profile scheme provides a way for programs or packages to store and retrieve small amounts of data separately for each user. This information would typically specify options chosen by the individual user.

Keywords

delete profile, list profile, merge profiles, options, read profile, write profile

Edinburgh Regional Computing Centre

James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh, EH9 3JZ. Telephone 031-667 1081

The Profile Scheme

The profile scheme provides a way for programs or packages to store and retrieve small amounts of data separately for each user. This information would typically specify options chosen by the individual user. For example, if the program were the text editor ECCE, the information stored might be whether letter case was to be ignored or not when searching for text strings, initial macro definitions, and monitoring level. It is stressed that different users can each select different settings.

Along with the information there is stored a version number. This relates to the format of the profile information stored for the program, and provides a mechanism for handling format changes in the information which the program stores. The details are given below.

The information is held for all programs using the profile scheme in a file in the user's process called SS#PROFILE.

Two external routines are involved:

```
%EXTERNAL %ROUTINE read profile(%STRING(11) key, %NAME info,
                                %INTEGER %NAME version, flag)

%EXTERNAL %ROUTINE write profile(%STRING(11) key, %NAME info,
                                 %INTEGER %NAME version, flag)
```

These require to be explicitly specified.

read profile reads the information stored under the specified keyword in file SS#PROFILE if it exists, and copies it to the %NAME-type parameter.

write profile copies the contents of the %NAME-type parameter to SS#PROFILE, creating the latter if necessary.

read profile parameters

The meanings of the parameters to read profile are as follows:

key — The keyword relating to the particular profile information; chosen by the program writer.

info — A scalar variable (not an array) of any type; usually a record or a string. read profile copies the information from SS#PROFILE to info. If the number of bytes held for the specified keyword in SS#PROFILE is greater than the size of info, then only as many bytes as info can hold are passed; truncation of the file information is from the right. If the number of bytes held in SS#PROFILE is less than the size of info (this includes the case where no information is held for the specified keyword) then the rightmost bytes of info are cleared to 0. See also flag, described below.

version — read profile returns the version number appropriate to the information currently stored in SS#PROFILE. This is 0 if no information is stored for the given keyword (or if SS#PROFILE does not exist).

flag — Set by read profile as follows:

0 Success.
1 Info held in SS#PROFILE is larger than size of info parameter passed. Some rightmost bytes of file info not transferred.
2 Info held in SS#PROFILE is smaller than size of info parameter passed. Some rightmost bytes of info parameter set to zero.
3 SS#PROFILE does not exist.
4 Keyword not found in SS#PROFILE.
5 Failed to connect SS#PROFILE. The attempts to connect SS#PROFILE are as follows. If the program is running in background mode, the program will attempt to connect SS#PROFILE a total of five times, with a delay of 20 seconds of cpu time between each attempt, before returning this flag. If the program is running in foreground mode, up to 3 attempts will be made, with a delay of 1 second of cpu time between each.
6 File SS#PROFILE corrupt.
7 Keyword is null.

If flag=2 on return then the info parameter has been partially padded with zeros, as described above. If flag>2 on return then the info parameter has been cleared to zeros and version has been set to 0. Note that only flag values of 5 or more indicate a definite fault.

write profile parameters

The meanings of the parameters to write profile are as follows:

key — The keyword relating to the particular profile information; chosen by the program writer.

info — A scalar variable (not an array) of any type; usually a record or a string. write profile copies the contents of info into SS#PROFILE, creating the file if necessary. See also flag, described below.

version — write profile sets the version number attached to the information which it is writing into SS#PROFILE to the specified value of version. If on entry version is negative, then the given keyword is deleted from SS#PROFILE. [Note that version is an %INTEGER %NAME parameter although its value is never changed by write profile. This is so that the formal parameter lists for read profile and write profile can be identical.]

flag — Set by write profile as follows:

0 Success.
1 SS#PROFILE was created.
2 Failed to create SS#PROFILE.
3 Failed to connect SS#PROFILE. First an attempt to connect it in write mode is made; if this fails another attempt is made after a delay of 1 cpu second. If this fails an attempt is made to connect it in read shared mode, and a flag of 3 is returned if this fails. If it succeeds, a copy of SS#PROFILE is made, and changed, then a NEWGEN is attempted; if this fails, a flag of 4 (see below) is returned.

4 Failed to create a copy of SS#PROFILE, or failed to NEWGEN it. A copy is made of SS#PROFILE if it can only be connected in read shared mode (see above) or needs to have its size altered. This usually entails increasing the size, but wasted space can be recovered during the operation and this may actually enable the file to be reduced in size.
5 SS#PROFILE already holds information for the maximum number of keywords permitted (ca. 500). Should not occur.
6 The info parameter is larger than 4060 bytes. This is the largest amount which may be stored for any one keyword.
7 Keyword is null.

No information has been written to SS#PROFILE if flag>1 on return.

Using the profile scheme

Without being prescriptive, I would envisage the scheme being used by a program as follows. On entry the program would first establish, by calling read profile, what information relevant to its operation was held. It would then respond accordingly, normally by setting up initial values for relevant variables. In addition, some means would be provided for the user to change the information held in the profile file; this could either be done from within the program itself or as a separate program.

One straightforward way of setting profile information from within the program itself is to follow the INITPARMS approach in Subsystem command OPTION: i.e. to provide a means for the user to specify that his defaults are to be the current settings of the relevant variables. Thus, in the case of ECCE, the user could set %L (case sensitive text location), %F (full monitoring) and some appropriate values for the macros %W, %X, %Y, %Z. Then if he gave the command %P say (P for profile), ECCE would call write profile to save %L, %F, and the current macro definitions. When the user called ECCE thereafter, this stored information would be read on entry and the program variables set accordingly.

The version number is provided at the suggestion of Sandy Shaw, who has also suggested an elegant use of it, as described below.

Suppose that a user used a program ZZZ three years ago, when version 2 of ZZZ's profile information was in use. Now, three years later, he starts using ZZZ again. His SS#PROFILE file contains information stored in the version 2 format, but now version 4 (say) of ZZZ's profile information is in use, which contains different information from that of version 2 and in a different format. The suggested approach is that when ZZZ makes its initial call on read profile it first checks that the flag returned is satisfactory, then uses the version number returned as a switch value and jumps to a piece of code:

```
vsn(0): ....
          :
vsn(1): ....
          :
vsn(2): ....
          :
vsn(3): ....
          :
vsn(4): ....
          :
```

The code following switch label 'vsn(1):', for example, transforms profile information held in the version 1 format into the version 2 format. It is assumed that a change in profile format will usually entail appending to the previous format, e.g. appending subfields to a record, although existing subfields could be ignored or reused. The conversion code is added to the program when version 2 of the profile format is introduced. It can also include an output message to the effect that a new version of the program was put into service on such and such a date, contains the following goodies ..., etc.

In the case of our user of program ZZZ, after the program has read the user's profile data a jump would be made to label 'vsn(2):' and the code following 'vsn(2):' and 'vsn(3):' would be executed. By this time the information returned by read profile would have been transformed into the version 4 format (the latest version). Thereafter, the program would call write profile to write out the information in the latest (version 4) format if what read profile returned earlier was not in the latest format. Then the run of ZZZ could proceed normally. The next time the user calls ZZZ, it will read a profile version number of 4, jump to 'vsn(4):' and proceed directly. Thus the code following each switch label is only executed once for each user.

This approach also provides a way of introducing the profile scheme into existing programs. When no information pertaining to a particular program is held in SS#PROFILE (or perhaps SS#PROFILE does not even exist), a version number of 0 is returned by read profile. A jump to 'vsn(0):' thus follows and the code there can set up the profile information, as described above. The version number of the new profile information will be 1. Just before the 'vsn(1):' code there will be a call of write profile, which will automatically create SS#PROFILE if it does not already exist. The code might have the following form for a program XXX:

```
      :
%RECORD %FORMAT prof f(%INTEGER a, b, %STRING(40) c)
%RECORD (prof f) prof
%CONSTANT %INTEGER program vsn = 1
%INTEGER flag, profile vsn
%SWITCH vsn(0:prof vsn)
      :
      :
read profile("XXXPROF" {keyword unique to this program},
         prof {returns information stored for "XXXPROF"},
         profile vsn {returns version no of stored profile info},
         flag {return values described above} )
%IF flag>4 %START
   printstring("Unable to access file SS#PROFILE.")
   newline
   %RETURN
%FINISH
-> vsn(profile vsn)

vsn(0):
   ! profile vsn was 0 on return - no profile info currently stored.
   prof_a = 4; prof_b = 7; prof_c = "Doughnuts"
   ! Elements of profile record set to defaults in use prior to
   ! introduction of profile scheme.
   printstring("XXX now uses a profile scheme - see documentation")
   newline
   ! This code is executed once for each user,
   ! so each gets message once only.

   ! The following two lines always precede the final 'vsn' label:
   profile vsn = program vsn
   write profile("XXXPROF", prof, profile vsn, flag)
vsn(1):
   ! Code to transform profile info to a version 2 format
   ! would go in here.

   ! Now transfer profile info to program variables.
   a = prof_a; b = prof_b; c = prof_c
      :
```

If the programmer decides later to change the profile information format, he changes the value of program vsn to 2, adds code after the 'vsn(1):' label to convert version 1 profile information to the new format, followed by the 'profile vsn = program vsn' and 'write profile(...' statements, followed by label 'vsn(2):'. The earlier code is not affected.

Changing the profile information format

Suppose that a programmer has made use of the scheme in some program. The relevant code might have the following form:

```
%CONSTANT %INTEGER program vsn=2
%SWITCH prof(0:program vsn)
%INTEGER profile vsn
%RECORD %FORMAT pform(%INTEGER a, b, c, d,
                      %INTEGER %ARRAY val(1:20))
%RECORD(pform) p
   :
   :
read profile(keyword, p, profile vsn, flag)
   :
->prof(profile vsn)
   :
prof(0): ! Set up a, b, c, val defaults.
   p_a = ......
   p_b = ......
   p_c = ......
   p_val(i) = .... %FOR i = 1,1,20

prof(1): ! New profile item d
   p_d = ......

   profile vsn = program vsn
   write profile(keyword, p, profile vsn, flag)

prof(2): ! Start of program proper.
   :
   :
```

Consider the above code. The program originally had items p_a, p_b, p_c and p_val in its profile record, and initialises these after label 'prof(0):'. Later, it was decided to add another item, d, to the profile record, and the record format pform was modified accordingly. The %CONSTANT %INTEGER program vsn was changed to 2 at this point.

Will this work? For someone using the program for the first time, the answer is yes: the code following label prof(0) will be executed, then the code following prof(1), and then the profile record will be written to the user's profile file.

But what about someone who used the program when version 1 was in service? The first time he uses version 2 of the program (i.e. in its above form) he will read into record p his version 1 profile information, which is 4 bytes shorter than p now is. And since p_d was introduced into the middle of the format, some of his profile information when read into the changed record will be misaligned.

In general it is unsatisfactory for the programmer to modify the profile record format (apart from appending new items to it), since this will cause trouble for users of earlier versions of the program, as exemplified above. However it is sometimes necessary to make radical changes to the profile information.

The following approach is suggested:

1) %RECORD %FORMAT p(%INTEGER a, b, c, d, %INTEGER %ARRAY val(1:20) %C
%OR %INTEGER %ARRAY x(1:24))

The record format has an alternative added to it consisting of a single integer array. (In some cases a byte integer array might be more convenient.)

2) Whenever the profile information is being initialised or manipulated into a new format, this should be done by reference to the p_x(..) array, not by reference to the variables p_a, p_b, p_c, p_d, p_val.

The above example would thus be changed as follows:

```
      :
      :
   prof(0): ! Set up a, b, c, val
      p_x(1) = .......; ! a
      p_x(2) = .......; ! b
      p_x(3) = .......; ! c
      p_x(i) = ....... %FOR i = 4,1,23; ! val

   prof(1): ! New profile item d.
      p_x(i) = p_x(i-1) %FOR i= 24,-1,5; ! Move array val up.
      p_x(4) = .......; ! d

      profile vsn = program vsn
      write profile(keyword, p, profile vsn, flag)

   prof(2): ! Start of program proper.
```

This method of handling profile information may seem more obscure, but it avoids problems later on.

Note the need to shift the array val within the profile information. It is better in general to put arrays within profile information at the start of the record, and also to leave some spare space after them in case they need to be expanded.

Direct amendment of profile files

This section describes three utility commands which are useful for manipulating the contents of files used by the profile scheme. Such files are usually named SS#PROFILE. The utilities were written by R.D. Eager of the Computing Laboratory, University of Kent at Canterbury.

Before the commands are used for the first time, access to them must be established by giving the following command:

Command: OPTION(SEARCHDIR=KNTLIB.GENERAL)

This need only be done once.

The LISTPROFILE command

This command lists the contents of a profile file. Its parameters are as follows:

filename — The name of the profile file whose contents are to be listed. This parameter defaults to SS#PROFILE.

output — The destination of the listing from the command. This may be a file or a device; the default setting is .OUT.

The output from LISTPROFILE consists of the key associated with the profile entry, the version number of the entry, and the actual profile information (in hexadecimal). Note that if the version number is negative, then the entry is ignored by the profile routines (see the description of 'version' under 'write profile').

The MERGEPROFILES command

This command merges the contents of two profile files. It takes three parameters, all of which are mandatory:

first input file — One of the two profile files to be merged.

second input file — The other profile file to be merged.

output file — The name of a file which is to contain the merged information from the two input files. It can be the same as one of the input files.

The command checks for the occurrence of a key in both input files; if this happens, the user is asked which one should be used.

The DELETEPROFILE command

This command deletes the information associated with a particular key from a profile file. Its parameters are as follows:

keyword — The keyword associated with the profile item to be deleted. If this parameter is omitted, the user is prompted for the keyword. This is useful if the keyword contains lower case letters, since these are usually converted to upper case when they occur in a command parameter.

filename — The name of the profile file from which the information is to be deleted. This parameter defaults to SS#PROFILE.