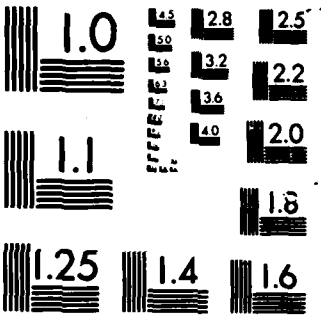
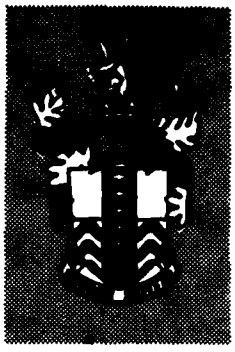
AD-A163 744 A PROCEDURAL IMPLEMENTATION OF PASCAL FILES ON FLEX(U) 1/1
ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND)
P D HAMMOND JUL 85 RSRE-MEMO-3883 DRIC-BR-90207

UNCLASSIFIED F/G 9/2 NL



UNLIMITED



RSRE
MEMORANDUM No. 3883

ROYAL SIGNALS & RADAR ESTABLISHMENT

A PROCEDURAL IMPLEMENTATION OF PASCAL FILES ON FLEX

Author: P D Hammond

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

UNLIMITED

(-A-)

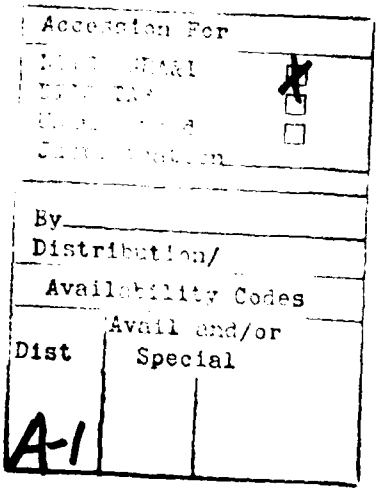
ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 3883

TITLE : A PROCEDURAL IMPLEMENTATION OF PASCAL FILES ON FLEX
AUTHOR : P. D. HAMMOND
DATE : JULY 1985

Abstract

This paper describes the implementation of Pascal files in an object-oriented manner using procedure values. A Pascal compiler has been written to the ISO standard for the Flex computer using these methods. Each file is represented by a package of procedures and every basic operation on the file is a call of one of these procedures. Such an implementation is only possible on a machine which supports true procedure values. Furthermore these procedure values must be able to exist in the filestore as well as in the mainstore of the machine.



0 Introduction

Inside a Pascal program, files and textfiles are manipulated by Pascal procedures such as **rewrite**, **reset**, **read** and **write**. Outside Pascal programs files have to be created and used by system programs or programs written in other languages. A Pascal file is no more than a sequence of values of some type, and a textfile is just a sequence of lines of characters. But the external system may have a more complex notion of the kinds of object that it can treat. For example, the editable files of Flex (1) have a structural concept of a paragraph, and Flex permits any structured values to be stored on backing store, not just simple sequences. We certainly wish to be able to read such files in Pascal programs, even though we may lose some of the information which is potentially available in them. We also wish to be able to write files from Pascal in various different external conventions, even though we can only create a subset of the possible structures.

A Pascal compiler has been written for Flex (2) to meet the ISO standard (5). This compiler represents files as a collection of procedure values which implement the basic operations on files. These procedures have certain relations to each other, for example, what has been written can later be read in the same order. These sets of procedures can be created externally to the Pascal program, so if one desires to read a Flex editable file a certain group of procedures can be generated which have a common non-local bound to them, namely the editable file which is to be read. In order to put files onto backing store in a way which simplifies their regeneration it is desirable to be able to store procedure values on backing store. Flex implements procedure values (3,4) and treats them as first class objects. This procedural implementation of Pascal files is based on an object-oriented approach to values.

The rest of the paper considers the specification needed for the procedures, and shows how they can be implemented. Each section consists of general discussion followed by details of the Flex implementation of Pascal files.

1 Specification of the package

A set of procedures is required which is capable of performing all the required Pascal file handling operations. The operations make use of a one element buffer which is implicitly part of all Pascal files. Briefly these operations are :-

rewrite - Clears the file and prepares it for writing.

reset - Prepares the file for reading.

put - Adds the value in the buffer to the data.

get - Puts the next element of the data into the buffer.

write - Adds a number of values to the data.

read - Extracts a number of elements from the data.

eof - Delivers TRUE if the end of the data has been reached.

Further operations only applicable to textfiles are mentioned later.

It would be possible to have a separate procedure for each operation but this is not strictly necessary since some operations are built up from simpler ones. For example, the operation **read(file,variable)** is equivalent to assigning the value in the buffer to the **variable** followed by the operation **get(file)**. It is sufficient to include a procedure which performs the operation **get** and to implement calls of **read** using this procedure. Similarly a procedure which performs the operation **put** can be used to implement calls of **write**.

The **file buffer** must be included as one element of the package since it can be used inside a Pascal program in the manner of a variable. Most of the procedures in the package need to access the buffer and this could be as a parameter or as a non-local. Since each file has a unique buffer, making this a non-local to the procedures ensures that the correct value is always used.

The operations **reset** and **rewrite** must be included as two separate procedures since they cannot be decomposed into simpler operations.

The Pascal function **eof** could be implemented by including a boolean variable as one of the elements of the package. Alternatively a procedure could be included which delivered a boolean so that a check could be made as to whether the file was undefined.

The package used to implement non-text Pascal files on Flex contains seven elements. These consist of one variable and six procedures whose purposes are :-

BUFFER - This is a variable of the same type as the elements of the file and is used as a one element buffer.

GET - This extracts the next element of the data and assigns it to the buffer.

PUT - This adds the value in the buffer to the data.

REWRITE - This clears the file and prepares it for writing.

RESET - This prepares the file for reading.

EOF - Delivers TRUE if the file is being written or the reading position is at the end of the file.

DELIVER - The purpose of this procedure will be given later.

These seven elements define a **file** as well as the interface between a Pascal program and the data held in the file. Moreover the system can only access the data in the file across the same interface. Thus the 'rules' of Pascal are extended beyond Pascal programs. The interface also hides details such as the actual representation of the data stored in the file.

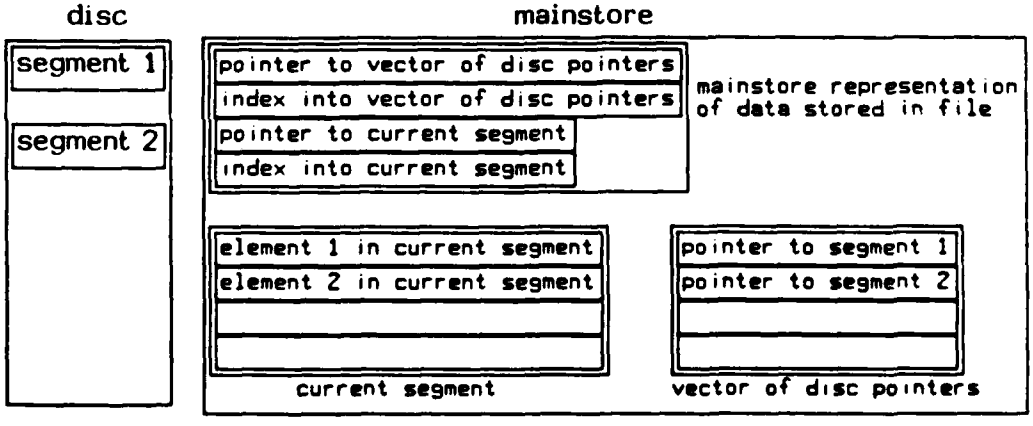
2 Data representation

The data held in a file consists of a sequence of values of the same type. These values could be held in mainstore, on a backing store or split between the two. This section briefly considers possible representations of these values.

A vector is a 1-dimensional array with a lower bound of one and could thus be used to hold a sequence of values. Another high-level structure which could be used is a list. These solutions require that all the values be held in mainstore and usually they are too extensive for this. If the file is being used to hold a small amount of data then keeping all of this in mainstore would prevent unnecessary interactions with backing store. However files are generally used to hold large amounts of data and consequently most of this must be in a backing store whilst the file is being accessed.

A common solution is to segment the data on backing store and this is the approach used on Flex. For convenience the backing store will be assumed to consist of a magnetic disc and the operation of storing values on the backing store will be referred to as writing to disc. The number of elements of the file in each segment depends on the size of one element. The top level representation of the data in mainstore is a set of disc pointers, the current segment and an index into that segment. When the file is being read the segments are brought into mainstore as they are required. Similarly, when the file is being written, completed segments are written to disc.

On Flex the disc pointers are stored in a vector and an index into that vector indicates which segment is currently in mainstore. The following diagram illustrates how the data is represented at a high level on Flex.



3 Method of implementation

Given the specification of a package which represents a Pascal file, a method of generating such packages is required. To perform their respective tasks the procedures of a package need to access certain values as non-locals, so these values are declared first. Then, using the formal specification of each procedure, code is written which makes all the necessary checks and manipulates the data as required. Finally the elements are put together to form the package.

The implementation of Pascal files on Flex will now be described as an illustration of the creation of such packages.

The procedures need to access the data held in the file and other values which indicate positions in the data and the state of the file. The common values used are :-

data - This is the actual data held in the file. It consists of a vector of disc pointers to segments and a vector of file elements which comprises the current segment.

indices - These are indices into the vectors which represent the data. There is an index into the vector of disc pointers and an index into the current segment.

state - This variable is used to determine whether the file is being read or being written. There is a third possibility, namely the state of a file immediately after it has been created. At this point the file is undefined and must, according to the ISO standard, be **rewritten** before it can be used. The procedures **put** , **get** and **eof** need to examine the state to check that it is correct for that file operation. Only the procedures **rewrite** and **reset** can change the state.

end_of_file - Although it can always be determined what the Pascal function **eof** should deliver it is more efficient to keep the information. The variable "end_of_file" holds the value that **eof** will deliver when called. Whilst the file is being written the function **eof** should always deliver TRUE so the procedure **rewrite** must assign TRUE to "end_of_file" and the procedure **put** doesn't affect it. The procedure **reset** makes "end_of_file" FALSE at the beginning of the reading process and **get** assigns TRUE to it when an attempt is made to read past the end of the data.

buffer - This is the **file buffer** and can hold one element of the file. As with other Pascal variables it is an error to use the value in the buffer if it is undefined. It is in this state initially, after calls of **put** and when the end of the data is read over. To detect this error additional information is required which may take the form of a tag which denotes whether the buffer is defined. For greater efficiency it was decided not to detect the use of undefined variables in the Flex Pascal compiler. Instead any variable that is undefined has a system determined null value. Similarly the value of the buffer after calls of **put** is this null value. The consequence of this method is that the value in the buffer is overwritten when the buffer becomes undefined and cannot be re-used.

rst - The first time that **reset** is called after writing to a file the data needs to be trimmed of unused elements. Trimming is the process of selecting a subset of values from a vector or array. The last segment of data is written to disc, the disc pointer to it is added to a vector and this vector is trimmed of unused elements. If **reset** is called again to permit reading to start from the beginning of the data then this tidying operation does not need to be done. The global boolean "rst" is used to determine whether the data has been tidied yet.

The elements of the package are now described in more detail.

BUFFER - Since the **file buffer** can be accessed independently it must be one of the elements of package as discussed above.

GET - The purpose of this procedure is to extract the next element from the data and assign it to the buffer. Checks are made that the file is currently being read and that the end of the data hasn't been reached. If the previous component read was the last one in the data then the boolean "end_of_file" is assigned the value TRUE so that further attempts to read from the file will fail. At the same time the value in the buffer is overwritten with a suitable null value. If the data is segmented then when the end of a segment is reached the next one is read into mainstore.

PUT - A call of this procedure fails if the file is not currently being written. Otherwise the value in the file buffer is added to the data and a null value assigned to the buffer. Complete segments are written to disc and the disc pointer obtained added to the top level representation.

RESET - This procedure is called prior to reading the file and only fails if the file is undefined. If this is the first call of **reset** since the file was written there is some tidying of the data required. The variable "rst" determines whether this is necessary. The variable "state" is changed to indicate that the file is being read and "end_of_file" is made FALSE. The relevant indices are set to point to the start of the data. A call of **get** places the first element into the buffer.

REWRITE - A call of this procedure will never fail. It erases any data held in the file and prepares it for writing. All indices are reset, "end_of_file" is assigned the value TRUE and "state" is changed to indicate that the file is being written.

EOF - This procedure is required to indicate when the end of the data has been reached in reading from the file. A failure occurs if this procedure is called when the file is undefined. After an attempt has been made to read over the end of the data TRUE will be delivered. Whilst the file is being written TRUE is always delivered.

4 Keeping the file on backing store

A file which is only a local variable in a program is called an internal file and the package discussed in the previous section is sufficient to implement it fully. External files exist outside Pascal programs and it must be possible to keep them on a backing store. In theory the whole package could be written to disc and certainly on Flex procedure values can be written to disc. But writing an existing mainstore procedure to disc is not always trivial and in this case the procedures have a complicated set of non-locals which makes it much more difficult. When a file is on disc, the ability to manipulate the data is not required, so only the data itself needs to be stored. It is also necessary to be able to recreate a mainstore file from the information on backing store.

Using the first six elements of the package, specified in section 1, the data held in the file can only be accessed by calling **get** and examining the buffer. This method would be an inefficient way of collecting the data together. The purpose of the seventh element of the package, the procedure **deliver**, is to provides the ability to write the data to disc and obtain the disc pointer to it more easily.

Having written the data stored in a file to disc it is necessary to be able to recreate the corresponding mainstore file. A new set of mainstore procedures needs to be generated which uses the data on disc as the initial value of the top level representation of the data. The other globals only need to be declared with suitable initial values. As the procedure bodies are dependent on the file type this process is not trivial but it is very similar to the original creation of the file. Instead of reproducing the code which creates a file of a given type one would like to use the code that was used to create the file originally. Thus when a new file is declared a procedure is generated which creates a mainstore Pascal file. This is used for the initial creation of the file and later to recreate the file in mainstore given a disc pointer to the data. It is this procedure which must be written to disc together with the data. This is the method used on Flex and details of the scheme are given below.

For any file type a procedure can be written which creates the corresponding mainstore file package. One of its parameters is a vector of disc pointers which corresponds to the data that the file will hold. For the initial creation of the file an empty vector is supplied as the actual parameter and later data can be read off disc to obtain suitable values. To facilitate the storage of the creation procedure on disc it has no non-locals. Thus any other values that are required inside the creation procedure must be supplied as parameters. In particular two system programs are needed which access the disc. One takes a value, writes it to disc and delivers back the disc pointer to it. The other takes a disc pointer, reads the value indicated and delivers the value.

Finally a disc pointer to the procedure itself is also required. This is necessary since the mainstore file must have the ability to produce a disc file which itself can reproduce the corresponding mainstore file.

The procedure **deliver** can now be given a specification :-

DELIVER - This writes the top level representation of the data to disc and delivers the disc pointer to it as well as the disc pointer to the creation procedure which is accessed as a non-local of **deliver**. If the procedure **reset** has not previously been called then the data may be in an untidy state. In this case the last segment of data must be written to disc and the pointer obtained added to the top level representation.

This procedure is special in that it is never called from inside a Pascal program. Instead a system program, which is used to write files to disc, selects the procedure **deliver** and calls it. The result of the system program call is the pair of disc pointers which the file on disc is composed of.

The sequence of events is thus as follows :-

Generate a PROCEDURE which constructs a particular mainstore file and write this PROCEDURE to disc.

Call this PROCEDURE with suitable parameters to create a mainstore file. The data will be an empty vector of disc pointers and the disc pointer to the PROCEDURE itself must also be supplied.

To store the file on disc **deliver** is selected and called. The two disc pointers obtained thus constitute the backing store version of the file.

When the file is required again in mainstore, the PROCEDURE is read from disc first. Then the outer level representation of the data is read from disc. Finally the PROCEDURE is called with parameters consisting of the data, the system disc programs and the PROCEDURE disc pointer itself.

For generality internal files have the same structure as external files and thus must also contain a procedure with the same specification as **deliver**. In this case the procedure will never be called and one with a dummy body is sufficient.

5 Textfiles

Pascal textfiles can be implemented in the same manner as a package of procedures although more elements are required. The additional file handling operations which the package must be able to perform are :-

eoln - Is used to determine whether the current position is at the end of a line.

page - Causes subsequent text to appear at the start of a new page.

write - Adds a textual representation of a list of values to the data.

writeln - Same as write but subsequent text must appear at the start of a new line.

read - Extracts a number of elements from the data converting back from the textual representation.

readln - Same as read but subsequent text is read from the start of the next line.

Again there are the same decisions to be made as to which operations can be implemented in terms of other operations. Both **write** and **writeln** involve the addition of strings of characters to the data which could be done by successive calls of **put**. Alternatively another element could be included in the package which added a string of characters to the data. A call of **writeln(textfile,x,y,z)** is equivalent to **write(textfile,x,y,z)** followed by by **writeln(textfile)**. Thus it is only necessary to provide an element in the package which performs **writeln(textfile)** to be able to implement **writeln**. Similar reasoning leads to the addition of an element which is equivalent to **readln(textfile)**.

A suitable set of additional package elements for textfiles is :-

EOLN - If the "state" of the file is undefined or "end_of_file" is TRUE then this procedure fails. Otherwise TRUE will be delivered if the file is being written or the reading position is at the end of a line.

WRITE - Providing that the file is being written this adds a string of characters to the current line. This is strictly unnecessary since **put** could be called on each character in the string. That would be inefficient because of the repeated procedure calls and their associated checks on the state of the file. This package element is used to implement calls of **write** and **writeln**.

WRITENEWLINE - This is used during the writing of the file to indicate that the end of a line has been reached. The current line is added to the current page and subsequent text begins on a new line. This is equivalent to a call of the Pascal procedure **writeln** with no parameters.

NEWPAGE - Used during the writing of a file to indicate that a new page should be taken. First the current line is added to the current page which is then added to the data. Subsequent text appears at the start of a new page. This is equivalent to the Pascal procedure **page**.

READNEWLINE - Used during the reading of a file to indicate that the reading should continue from the start of the next line. This is equivalent to a call of the Pascal procedure **readln** with no parameters.

It is desirable for a textfile on disc to be a Flex editable file to facilitate examination and editing. Thus the procedure **deliver** writes the characters in the file to disc in the form of an editable file. A system program is provided which calls **deliver** and thus converts a textfile into an editable file. Also provided on Flex are system procedures to generate blank or empty textfiles and to convert editable files into textfiles.

6 Conclusions

A method has been outlined which enables Pascal files to be implemented as a package of procedures. Moreover details of the Flex implementation of Pascal files have been given as an illustrative example.

One of the strong points of this sort of implementation is the added integrity it provides. The data can only be accessed across the interface of the package both in Pascal programs and by the system.

An example of the benefits of such an implementation is the ease with which the standard Pascal I/O can be included. A set of procedures can be generated whose bodies are specific to the task of a standard output channel. This may send the output to a screen or a local printer for instance. This package has the same specification as any other textfile and is used in a Pascal program as the standard output textfile. On Flex if the standard Pascal output channel is used then it corresponds to a formal program parameter and the user can supply any textfile for this purpose. A system program is available to generate a pair of textfiles which can be used for interactive input/output on the screen.

Another example of files which have the same specification as normal files but which behave differently are those which do not write data to disc whilst the file is being used. These are provided as an extension to ISO Pascal files on Flex and all the data held in the file is kept in mainstore. This allows the programmer to use a 'small' file more efficiently by preventing excessive disc interactions. These files are easily implemented by making a few changes to the bodies of **put**, **get**, **reset** and **rewrite**.

7 Acknowledgements

I wish to thank Paul Taylor and Kim Curtis who constituted the rest of the team which wrote the Pascal compiler on Flex. I also wish to thank them, Dr. Foster and everyone else who made helpful comments on the content and style of this paper.

8 References

1. I.F.Currie, P.W.Edwards and J.M.Foster " Flex firmware", RSRE Report No 81009 (1981).

2. K.Curtis, P.D.Hammond and P.D.Taylor "A Pascal compiler for Flex" (in preparation)

3. J.M.Foster, I.F.Currie, P.W.Edwards "Flex : a working computer architecture based on procedure values", RSRE Memorandum No 3500 (1982)

4. I.F.Currie "In praise of procedures", RSRE Memorandum No 3499 (1982)

5. "Specification for the Computer language Pascal", BS 6192 (ISO 7185) (BSI 1982)

DOCUMENT CONTROL SHEET

Overall security classification of sheet ..... UNCLASSIFIED .....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference MEMORANDUM 3883	3. Agency Reference	4. Report Security Classification U/C
5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location ROYAL SIGNALS AND RADAR ESTABLISHMENT		
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location		

7. Title

A PROCEDURAL IMPLEMENTATION OF PASCAL FILES ON FLEX

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
HAMMOND, P				

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

UNLIMITED

Descriptors (or keywords)

continue on separate piece of paper

Abstract

END

FILMED

3-86

DTIC