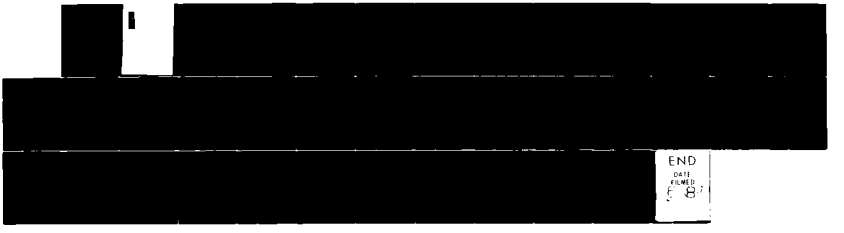
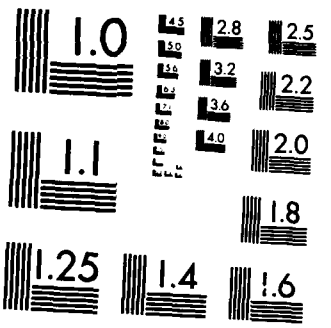
AD-A181 035 AN EVALUATION OF THE FLEX PROGRAMMING SUPPORT ENVIRONMENT(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) M STANLEY AUG 86 RSRE-86003 1/1

UNCLASSIFIED DRIC-BR-101371 F/G 12/5 NL



END
DATE FILMED
5 87



Royal Signals and Radar Establishment

Report 86003

AUTHOR: Margaret Stanley.
DATE: Aug 1986

An evaluation of the FLEX Programming Support Environment

This paper gives a personal reaction to the Flex Programming support environment, developed at RSRE, Malvern. It discusses the new approach to program development necessary to take advantage of the environment, and evaluates the environment against some criteria for programming environments in general.

An evaluation of the FLEX Programming Support Environment

Contents

An evaluation of the FLEX Programming Support Environment

1. Introduction

This is a personal reaction to the Flex programming support environment (PSE) developed at RSRE, Malvern. The paper discusses the features needed for programming support and evaluates how well the Flex PSE satisfies these needs and how it avoids some problems familiar to users of conventional systems.

Flex[1,2,3, and 4] is a multi-language PSE intended to be used for developing programs and software prototypes and for testing new ideas on software methods and designs. It demonstrates the implementation and use of some ideas for improving productivity both in program development and maintenance. It provides users with a large amount of software for programmer support but does not currently provide tools for host/target software development nor for phases of the software development cycle other than programming, such as tools for project management or software design. Such tools could be developed.

Flex is noticeably different from other PSEs and it took some effort to learn to use it effectively. On first encountering it I was sceptical - yet another operating system to learn, with its own peculiarities. It would no doubt have a lot of positive features offset by an equal weight of disadvantages. I have been pleasantly surprised. Experienced Flex users show a similar enthusiasm to that of Unix devotees but with more justification. The PSE is built on the Flex capability object oriented architecture developed at RSRE, Malvern. Development since the architecture came into use in 1978 has been mainly a response to requests from programmers. The result is a highly interactive and usable PSE that encourages software re-use. The software base provides all normal operating system facilities, including compilers for Algol68 and Pascal. Compilers are under development for Ada(*) and ML.

The Flex capability computer architecture has (so far) been implemented in microcode on 4 hardware configurations including one multi-user implementation in which 3 Flex computers share a common filestore and common peripherals. The most recent implementation is on the ICL Perq. Implementation of Flex on existing computer systems (without re-microcoding) is being considered. Implementation of the Flex PSE on different hardware requires different user interfaces, particularly an editor tailored to the type of vdu available on the system. However, the underlying Flex architecture and the philosophy of the Flex PSE persist across all implementations.

Before giving further details of Flex, some criteria for evaluating a PSE are listed (section 2). The technical presentation of Flex (section

*Ada is a registered trademark of the US DoD.

3) describes those features that are important to understanding it as a PSE. The effect on the method of use and on the acceptability of the PSE is discussed. An evaluation of Flex (section 4) against the criteria in section 2 precedes a summary (section 5) of the characteristics that make Flex different as a PSE. The paper ends (section 6) with some general reactions to the Flex PSE and ideas for its future.

2. Criteria for evaluating a PSE (Programming Support Environment)

A programming support environment aims to support the evolution of programs. This includes support not only for writing and testing individual programs and for integrating sets of programs to allow them to work effectively together, but also support for the process of change and for configuration control of programs and sets of programs.

Summarising the requirements suggested in references 5 and 6, a PSE needs the following major characteristics.

1. A PSE must be trustworthy. Programmers should not need to guard against possible corruption of filestore objects caused by hardware failure, system failure, simultaneous access by more than one user or misuse of filestore addresses. Similarly programmers should be able to trust the system to prevent corruption of or illegal access to values by misuse of mainstore addresses.

2. A PSE needs flexible access controls that allow users to impose different controls according to circumstances. They should be powerful enough to prevent illegal or unauthorised use of filestore or of mainstore values.

3. The need for privilege should be minimised. Privilege weakens integrity by making it possible to break normal access or integrity controls. It should be unnecessary to invoke privilege to achieve reasonable objectives such as enabling code sharing between different users of a program or calling a utility from a user program.

4. A PSE should provide good support for program evolution including support for testing programs, diagnosing program errors and amending programs. Support is also needed for administrative functions such as keeping track of relationships between parts of an evolving software system, recording and controlling changes and propagating authorised change.

5. A PSE should support the use of data structures, allowing users to create explicit data structures and to use them when data is passed between programs or preserved on filestore. Programs contain structured data. The structure and information about structure should be preserved when values are delivered from programs. There should be protection against mis-use of structured data.

6. A PSE must be capable of growth. It should be easy to add new tools and new data structures without affecting existing tools and data structures.

7. A PSE should minimise programmer effort in writing new software. Programmers should be encouraged to re-use existing tools, programs or parts of existing (granular) tools in a new context. Where possible standard methods should be supported for example for exception handling, for interfacing to user terminals and for checking that data input to programs has the expected structure.

8. The user interface should be simple to use, consistent and orthogonal. To avoid user confusion tools should be accessible from anywhere in the system (not just from the outermost (operating system) level). There should be encouragement to make new tools interface with the user in the same way as existing tools.

9. A PSE should protect the user (as far as possible) from the consequences of his own mistakes. There should be guards against common user errors, such as calling a program with invalid parameters, premature deletion of something from filestore or accessing the wrong value because a name has changed its meaning (perhaps as a result of a program amendment).

10. Adequate help should be easily accessible at all times. A PSE needs clear, well structured documentation, which can easily be updated in line with changes to the system (e.g. incorporation of new tools). A "help" facility that applies to every tool and to the PSE as a whole should be available from anywhere in the system.

11. A PSE should support the definition and use of relationships between different types of object. Users need to be able to search the filestore for related objects (such as the design documentation associated with a program or the source text used in a given version of a procedure or program). This may imply a requirement for a database underpinning the system (perhaps more necessary for project support than for programming support).

3. The underlying philosophy of the Flex PSE

In order to see how Flex meets these criteria it is necessary to understand how and why Flex is different from a conventional computer system. This section discusses some of the novel concepts on which Flex is based and the effect they have on the process of program evolution.

3.1 Filestore updating

Values on filestore (often called files) are values retained between user sessions. They are often large values spanning several

non-contiguous blocks of filestore. Operating system software presents users with files rather than with blocks of filestore. A corrupted or inconsistent filestore is one in which the operating system presents the user either with no value for an existing file or (worse) with a wrong value.

One cause of filestore corruption on conventional systems is partially updated filestore resulting from a hardware or system crash during file updating. A great deal of software effort is devoted to protecting systems from a partial update, attempting to detect filestore inconsistency and recovering a consistent filestore. Although such software is normally provided by the operating system, programmers cannot afford to ignore the possiblity of filestore inconsistency, since standard operating system checks cannot be expected to detect internal inconsistency within partially updated files.

The Flex architecture is designed to prevent the possibility of partially updating filestore. Flex filestore is non-overwriting[13] apart from a few root pointers which are updated as a single atomic action. There is no facility to alter an existing filestore value (other than a root pointer). A user can either write something to filestore or he can read or execute something that has already been written. He cannot alter the content or size of an allocated block of filestore. The architecture ensures that a root pointer is a single word block that can be updated only as a unitary operation. The root pointers usually contain a reference to a dictionary (a set of associations of names with filestore values) from which other allocated blocks can be reached. Even dictionaries are not updated. The only way to change a dictionary is to create a new one (which is an edited version of the old dictionary) and then update the root pointer to point to the new dictionary.

Since the only operation that can overwrite anything on disc is unitary, partial overwriting is impossible. Successive values of a root pointer each give access to a completely consistent set of unchangeable filestore values. If the system crashes at any time the user will either have the old value of the root or the new value. He cannot have a partially updated value. Since partially updated filestore is impossible the programmer need not provide software to detect or recover from partial updates.

3.2 Procedures and the command interpreter

Flex treats procedures as first class, context independent values or objects[2,8,9]. A procedure is a true procedure value in the sense of Landin[7]. Procedure code needs a context (its non-locals) to make it executable and on Flex the context or environment in which the procedure runs is bound in as a part of the procedure value. One consequence is that, unlike conventional operating systems, the Flex command interpreter, curt[4], can call procedures directly without

requiring that they first be formed into a program. Curt is an interactive procedure which implements a very simple interpretive language called Curt that allows users to call procedures. The effect of calling a procedure from curt is the same as that of calling it from a user procedure. The result is delivered to curt (which delivers it to the user via the screen editor), in the same way as the result is delivered to any other calling procedure at the point of call. All tools or programs in the PSE, whether system provided or user provided, are just procedures. There is no distinction on Flex between calling a procedure and invoking a program or tool. Any procedure (including a utility) can be invoked equally easily from a user procedure or from curt.

3.3 Flex modes and values

Strong data typing is an accepted feature of high level programming languages. Programmers have the ability to define a variety of data types to suit a particular problem and appropriate use of values of each type within a program is enforced. Flex has a mode system[11] to provide at the operating system level the equivalent of data typing in high level languages. Without such a system the structural information defined within programs is not preserved outside the programs, when data is passed between programs, stored on filestore or handled by an operating system.

Every object or value handled on Flex, whether on filestore or in mainstore, has an associated Flex mode (value type) that is used to indicate how the value is to be interpreted, and the operations that are valid on it. Values vary in type from simple integers to much more complicated structures. Atomic modes are used to describe values whose structure need not be visible to the user.

In contrast to the limited set of filestore types available in most computer systems the variety of Flex modes is boundless. Flex architecture supports the use of values of any size whether in filestore or in mainstore. Most mainstore objects (including procedures) have direct analogues in Flex filestore. The ability to keep values of such a wide variety of Flex modes on filestore gives much greater flexibility than encountered with conventional filestore.

The basic modes provided by Flex include analogues of all the modes and data types encountered in modern programming languages as well as some modes peculiar to Flex. The user can also invent additional modes (that are composed from the basic modes) to suit his problem. The additional modes may be atomic or they may have explicit visible structure according to user requirements.

For example, some of the basic modes are:

Int (the value is an integer);
Char (the value is a character).

There are also mode constructors, to permit more complex modes to be defined. For example

(Int,Real,Bool) (the value is a structure containing an integer, a real number and a boolean value);
Vec Int (the value is an ordered set (a vector) of integers);
Vec Vec X (the value is a vector of vectors of values of some mode X).

In addition there are modes, peculiar to Flex, some of which are discussed below.

3.4 Some special Flex modes

Capability modes
Those values in Flex which require some degree of access control to preserve system integrity are represented by capabilities. The Flex microcode (which is fixed and inaccessible to users, being used as an extension to the hardware) ensures that a value represented by a capability can be used only in the way authorised by the capability and only by a holder of the capability. A capability can be created and modified only by the Flex microcode. The capability mechanism, which is not susceptible to interference from the software, controls access to mainstore values (such as procedures), filestore values and remote values (values on other Flex computers). This ensures that only operations of the right kind are applied to mainstore and filestore values and only legally accessible values can be reached. An important and unusual aspect of Flex capabilities is that they can be treated like other values in that they can be held anywhere on filestore or in mainstore, used in programs and can be passed to another user, giving the other user access to the controlled value.

In a sense a capability is a pointer created on behalf of the user by the microcode, but the capability also holds the size of the value and information on the type of use (read only; read/write; execute) permitted by the microcode. A capability is not an address. It is rather a permission to use something (such as a region of store or a remote facility) in an authorised way. This means that users are unable to compromise the integrity of the system by unauthorised use of mainstore or filestore addresses. They have no meaning on Flex. Since capabilities cannot be modified, the type of access permission that they allow cannot be changed. A user who wishes for a different type of access to an object must request a new capability, which can only be granted by the microcode. For example, a user with a read only capability to a block of mainstore cannot manipulate it to obtain

read/write access. A new capability may give different access rights to the same value. Similarly, a user with a capability to execute a procedure can execute the procedure but he cannot do anything else to it, such as reading any internal values.

The mode of a capability value indicates the mode of the value to which it gives access. For example:

Edfile (the value is a capability to read a file (an edfile) that can be handled by the editor);

Module (the value is a capability for an object (a module) that gives access both to compiled code and to the source text from which it was derived. A module is the unit of separate compilation on Flex);

Vec Edfile (the value is a vector of capabilities to read editable files);

Ptr X (the value is a mainstore capability for a value of mode X).

The Flex architecture ensures that the only software or user operations involving a capability are:

1. request a new capability from the microcode (any procedure can do this; it is not a privileged operation);
2. store a capability for future use;
3. copy the capability to another user;
4. delete a copy of the capability (this does not delete the value to which the capability gives access, another copy of the capability may still exist);
5. use the capability as authorised by the microcode.

An example may clarify the concept of capabilities. Suppose that a user wishes to create a text file on filestore. He has a capability to execute the editor which gets a read/write capability for a region of mainstore, in which it holds the text being edited. When the editor is ready to write the text to filestore it requests a capability to create a read only block of appropriate size on filestore. It writes the text, using the create block capability, and gets back a read only capability for the block of text, which it passes to the user. This capability can be used only to read the text, and the text can be accessed only by using this capability. A user can pass the capability to another user, thus giving someone else read access to the text, or he can relinquish the capability if he no longer wishes to have access to the text. No other use can be made of that area of filestore as long as the capability exists, so the filestore cannot be over-written and then the old capability used to gain access to new data. The microcode takes care of this.

Procedure modes

Because every procedure (or program) on Flex is a value like any other value, every procedure has an associated Flex mode. The value of a procedure is a capability to execute the procedure and its Flex mode is a triple written as: input_parameter_mode -> result_mode . The

symbol -> is one of the elements of the triple and shows that the value is a procedure. A procedure to count the entries in a vector of integers is of mode: Vec Int -> Int .

There is an empty value on Flex whose mode is Void. A procedure on Flex that apparently delivers no result actually delivers the empty value. Its mode is written: input_parameter_mode -> Void . For example, a procedure to list an edfile on a printer has mode: Edfile -> Void and a procedure that takes no input parameters has mode: Void -> result_mode .

Although a procedure cannot itself be a filestore object there is an analogous filestore object called a filed procedure, of Flex mode: Filed (procedure mode) . It can be called from curt in the same way as an ordinary procedure.

When data is passed from one program to another, each program must correctly understand the structure of the interface data. On Flex the expected structure is explicit. Before calling a procedure curt checks the Flex mode of the input parameters. Any attempt to apply a procedure to a value of the wrong Flex mode fails, so users can be confident that procedures will not be applied to unexpected data structures. This is in contrast to conventional systems that cannot explicitly define or handle the wide variety of structures needed. Flex can. Flex therefore avoids the unanticipated or undefined results which can result from misinterpreting unstructured data. This valuable form of protection greatly reduces the amount of data checking needed within user procedures.

A Flex mode is advisory. It is information to the command interpreter and to other procedures on how to interpret a value. It can legitimately be changed, and facilities are provided to do this. A user can select a mode that accurately reflects the use to which the value is to be put. A mode change does not change the value. If a value is given a mode which cannot match the value it will be rejected when curt attempts to interpret the value. The utility that creates a procedure can select the Flex mode for the procedure that matches the external interface of the procedure as defined by its source text. However, since Flex modes are richer in scope than the modes available in most programming languages, in particular in the ability to distinguish a capability value from other pointers or integers, a user may prefer to define the procedure mode for himself. For example he may wish to use the procedure mode to protect a procedure that expects a capability from being called with an integer parameter even if the programming language treats the capability as an integer. System integrity does not rely on correct modes since only the interpretation of the value is changed. A mode change cannot be used to create a capability from another value nor to modify the rights conferred by a capability.

The possession of a procedure capability allows the holder only to

execute the procedure. It does not allow him to dismember the procedure to find how it works, what other procedures it might use or the values of its non-locals. This is the basis of much of the security of data in Flex filestore and mainstore. You cannot dismember a procedure by giving it a different Flex mode.

Flex modes Mode and Moded

A mode can itself be regarded as a value (of Flex mode Mode) that can be passed into and out of a procedure. For example a procedure to convert a mode to a character string is of mode: Mode -> Vec Char.

Sometimes a procedure may need to examine the Flex mode of its input. Consider a procedure intended to handle correctly a value which may be of any mode. A special mode, called Moded is available. A value of any mode can be expressed as a Moded value. The representation of the Moded value contains both the value and its mode. Thus the mode of a value can be passed into and out of procedures together with the value. Curt can, when appropriate, convert a value V of mode M into a Moded holding both V and M, and it can untangle a Moded to deliver it as a value of the correct mode (i.e. as a value V of mode M).

For example, consider a procedure "show" that takes any Flex value and displays it on the screen in a form appropriate to its mode. The Flex mode of "show" is: Moded -> Edfile . The procedure "show" untangles the Moded (V, M) and displays V in appropriate form. If "show" is called from curt the input value may be of any mode, because curt will convert it to Moded form.

Consider a procedure "find" which takes a name and searches a dictionary for a value associated with the name. The value may be of any mode. The procedure delivers the value together with its mode, as a Moded value. The Flex mode of "find" is: Vec Char -> Moded . Curt will untangle the Moded and deliver the value in the correct mode.

One result of embedding the mode information with the value in the Moded mode is that a single procedure can handle a boundless variety of data structures instead of needing a different procedure for each different structure. This reduces the quantity of software needed to handle different data structures, and also allows existing functions to handle new modes created since the function was written.

Exceptions

Sometimes a prccedure fails, because of an internal error (such as dividing by zero), an explicit failure or an Ada exception. When this happens, instead of returning a value of the expected type to the calling procedure an exception value is returned giving access to information on the failed procedure and its values at the time of failure. The calling procedure can handle the exception internally or it can itself fail, adding information on its own values to the exception value that it passes to its own calling procedure. Exception values thus

accumulate information as they are passed out through the calling procedures to the command interpreter (as a value of Flex mode Exception).

A procedure "diagnose", of mode Exception ->Void can be called from curt or from a user procedure to present to the user information on the state of the procedure at the point of failure. The information presented includes a list of procedures called before reaching the failure point, the names and values (displayed in appropriate format using procedure show) of all local variables in each called procedure and the source text of each procedure with a marker at the failure point. It is important to note that the source text presented to the user by diagnose is the actual text from which the procedure was derived. The user can browse through the information at the terminal or, if he wishes, he can invoke curt (from within diagnose) to call other procedures on a value presented by the diagnose procedure. Thus the full facilities of Flex are available to allow the user to examine further the reason for the failure. This is a very powerful tool which allows a user to home in very quickly on the source of his errors. It is unnecessary to abandon the diagnosis in order to look at machine held documentation, to examine source text or to experiment with values in the failed procedure. Unlike some diagnostic tools, there is no need to insert trace statements in the source text, nor to use conditional compilation.

The ability of Flex to handle exceptions as normal values allows them to be treated in a uniform way. This reduces the amount of special software needed in an application that caters explicitly for exceptions as well as providing necessary information to the diagnostic procedure.

The owner of a procedure issued to other users may wish to prevent the use of "diagnose" or similar tools for unauthorised examination of the internal working of a failed procedure. Programmers have the ability to exclude information from the exception value, as discussed later (see section on access control).

3.5 The user interface to Flex

Edfiles

Flex is designed for interactive use. All interactions take place via the editor, which is used (often implicitly) for all screen based input/output. In addition to the rich set of text editing abilities, with screen handling facilities for multiple windows and arbitrary pictures, the editor manipulates values included in the text. Curt is fully integrated with the editor allowing the user to produce and use such values. An editable file (an edfile) is a database-like object that may contain a mixture of characters, other edfiles, modules, integers, filed procedures etc. Since any edfile may contain others, edfiles may be a structure of edfiles and other values, with the restriction (enforced by the Flex filestore) that the structure must be acyclic.

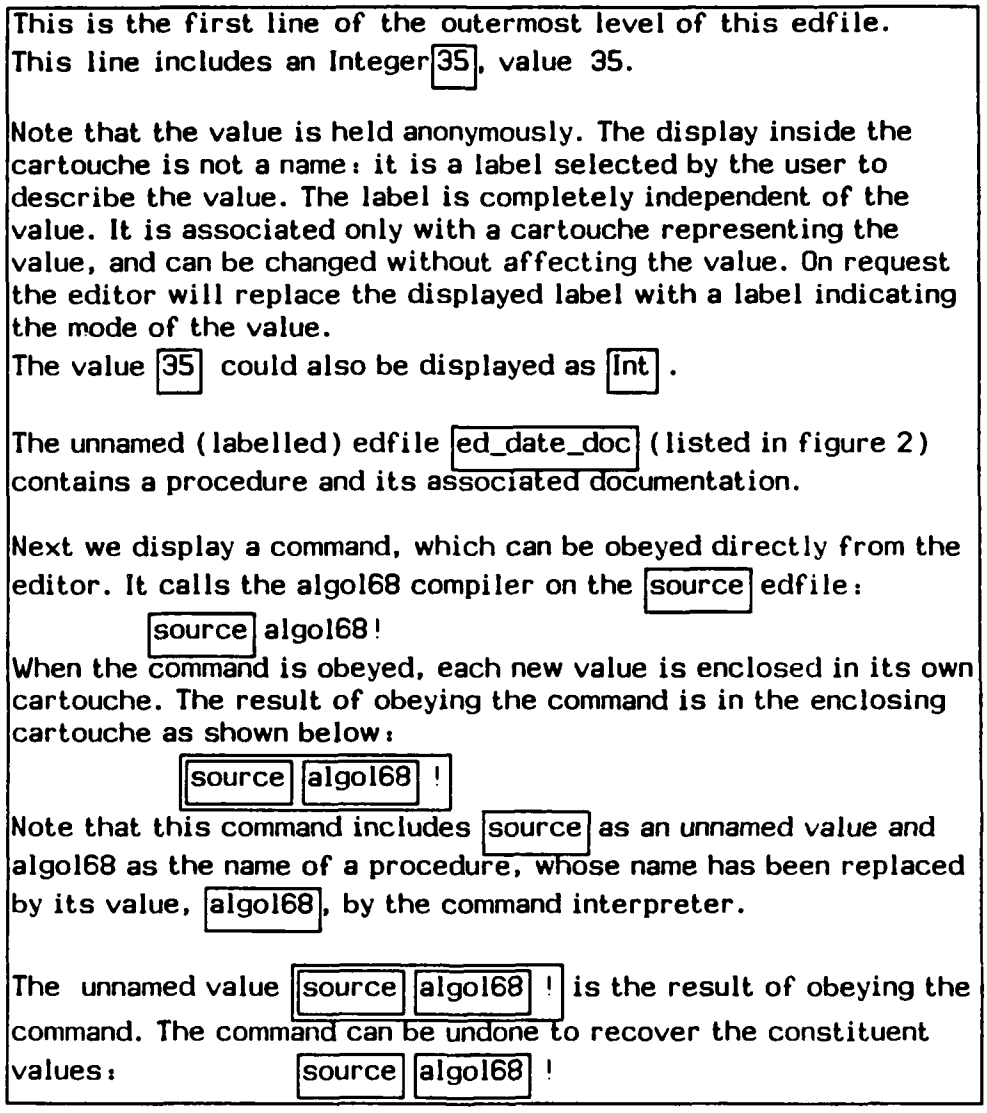


Figure 1. Edfile example.

Figure 1 is a trivial example of the content of a structured edfile as displayed by the Flex editor. The cartouches (boxes) represent values of any mode. Anything not enclosed in a cartouche is normal text. The structure shown in figures 1 and 2 is tree-like. It is not strictly speaking a tree because the same value may appear in more than one place in the acyclic structure. Every occurrence of a given capability in the tree-like structures gives access to the same value no matter what is displayed in the cartouche. Only a single copy of the value exists. The editor provides facilities for finding out about the value represented by a cartouche. The mode of the value dictates how the information will be given. For example if the value is a structure of numbers the editor will display the structure and the numbers, if it is a

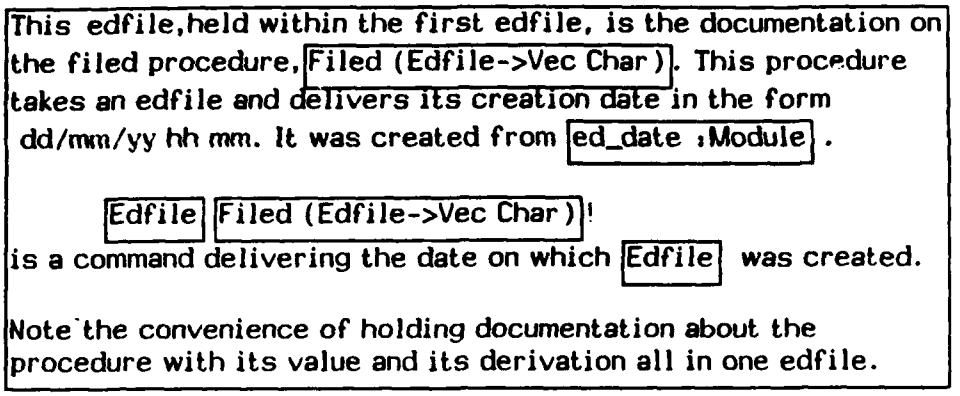


Figure 2. Edfile labelled ed_date_doc.

character string it will display the characters, if it is an edfile the editor will display its content.

When updating an edfile or any other value it is unnecessary to take a back-up copy to guard against unwanted changes; filestore is non-overwriting so filestore objects cannot change underneath you. An apparent change to any filestore value involves getting a capability for a new value. The old (unchanged) value continues to exist until all copies of the capability for it have been deleted. Commands can be undone to recover old values, as shown in figure 1. An apparent change to an editable file actually gives a new capability for the edited file. At this stage both the old and the new (changed) files exist as separate values. A user may have capabilities for both. He may choose to retain both capabilities or to delete his copy of the capability that he no longer wants. The old file has not changed and holders of a capability for it will still be able to access it. The user has simply gained access to another editable file.

Values and names

Names on Flex do not have the importance that they assume on most other PSEs. Most values are never named[10]. As shown in figures 1 and 2, they are created without names (as a result of a procedure call), held anonymously in structured edfiles and used without names in curt commands.

A dictionary on Flex is a set of name/value associations. If a name is used for a value curt finds the corresponding value in the dictionaries accessible to the user. Names may be temporary or persistent. Persistent names persist from session to session and therefore apply only to filestore values. A user normally has access to two dictionaries of persistent names, his private dictionary for values named by him and the common dictionary for named values shared by all Flex users. When users are created they can have any number of dictionaries associated with them. Dictionaries can be shared between individuals.

Temporary names do not persist between sessions and may therefore be associated with values of any Flex mode, including mainstore values. Temporary names are useful for objects needed frequently during a session, and which contain mainstore capabilities, such as a vector of objects for testing a procedure. Temporary names are usually held in a temporary user dictionary in mainstore but temporary name/value associations can also be set up within a user procedure, to apply only to calls to curt made within that procedure. This enables a user to write procedures that use curt to interact with the rest of the Flex PSE while using their own name space, that does not apply outside the procedures.

Curt commands use both names and unnamed values. Unnamed values are held with text in edfiles, and are accessed (using the editor) via their position in the edfile. In practice only those values to which a user wishes to refer in a wide variety of different contexts are named, plus one outermost edfile that contains the unnamed values. Private dictionaries tend to be very small. Typically there will be one named edfile for each major piece of work of interest to a user. The grouping of objects within a single structured edfile is purely a matter of convenience, a way of keeping related things together. Documentation can be naturally structured with each chapter in a different sub-edfile, which in turn contains sub-edfiles for sections.

All names in a Flex dictionary are in scope simultaneously because the dictionaries are not tree structured. They are usually so small that a tree structure is unnecessary. Tree structured directories of name/value associations can cause confusion. In many conventional environments the naming system allows users to navigate from a root directory to a subdirectory but within the subdirectory only the local name-space applies. A name is in scope only in the appropiate subdirectory and the same name occuring in different subdirectories will be associated with different values. Navigation through the tree-like structure of an edfile to seek an unnamed value does not affect the name-space within which a user is working. In addition to the usual functions of text editing, the Flex editor allows the user to navigate to the unnamed values. After the editor has been called on a named edfile the editor displays a window on the outermost level of that file. To proceed down the tree-like structure to the next level in the tree, a user calls the editor recursively on an edfile in the current level of the tree. He proceeds until he reaches the position in the tree-like structure containing the object he seeks. This is usually a very rapid process, because the window selected for display at each call of the editor includes the cursor at its position when that edfile was last used. (In fact the position when it was created, since edfiles are not modified, only re-created).

Initially the concept of not naming things led to a feeling of unreality, and to a fear that it would be difficult to find an anonymous file. However, it soon became apparent that labelled values surrounded by

explanatory text are easier to identify correctly than if the user must rely only on names, however well structured the dictionary hierarchy and however clear the naming convention. It is difficult to keep track of an ever increasing dictionary of artificial file names. The Flex editor can be used to move a capability to a different place, or to restructure the hierarchy, so a badly structured hierarchy need not be a persistent problem. Since capabilities can be repeated, it is possible to have an object (anonymous or named) displayed with helpful labels in more than one hierarchy.

Machine held documentation

Flex documentation is held on Flex in structured edfiles which a user can browse through simply by calling the editor. There are no printed manuals associated with Flex. A tutorial is held on the machine to guide the new user into the new ideas and to demonstrate how to start developing a program. A shared documentation edfile, structured according to topic, holds not only descriptive text but also the values of procedures, modes and modules (units of separate compilation) accessible to all users. Values are held with their documentation as in the edfile example given earlier. When new procedures or modules are made available for general use they are added to the shared documentation file with their documentation. In addition there is an "info" procedure that delivers an edfile of information about any named value. The quality of the information provided is usually adequate. The user can provide an information file (which can then be accessed using info) about any object held in his own dictionary.

It is, of course, possible to print out any or all of the machine held documentation, but once a user is familiar with the structure of the documentation files, it is usually quicker and easier to access the information on-line. Although it is not always easy tc find information, particularly if unsure which headings it comes under, it is no worse than being unable to find the required information in a set of printed manuals (and it wastes considerably less paper!).

Most conventional systems now have a help facility that covers the basic operating system and utilities, but help with user supplied tools is not always available via the general help mechanism. Tools are usually expected to provide help from within themselves. Help is sometimes available only at the outer (operating system) level. Flex users can examine any of the documentation without abandoning the current task. For example, when a user needs to check the specification of a module when writing some source text, he examines the relevant documentation and returns to the position in his source text where the problem arose.

Flex reduces the problem of maintaining consistency of information held (for convenience of access) in several different places by holding a capability for an edfile containing the information where needed instead of repeating the information. There is still a problem when correcting

documentation errors in that replacing one instance of a capability with a new capability does not automatically correct other instances of the capability. The system is being extended to include updateable editable values.

3.6 Procedure values

Information hiding

One consequence of treating a procedure as a value, with its non-locals bound into it, is that values may be concealed in a procedure. Procedures may be parameters for other procedures or may be delivered by other procedures (provided the language also supports this notion, as does Algol68). Values (both local and non-local) and input parameters of the delivering procedure can be bound into the delivered procedure, and thus hidden from a caller of the delivered procedure.

For example, consider a procedure, make_channel, that creates a channel for passing messages together with the procedures to access the channel. One channel is created by each call of make_channel, and that channel can be accessed only by using the procedures delivered by that call. Make_channel takes an integer giving the size of channel (i.e. the number of messages (vectors of characters) it can hold) and delivers three channel access procedures. The Flex mode of make_channel is:

```
Int -> ( ( Vec Char -> Void) , ( Void -> Vec Char ),(Void->Void) )
```

The first delivered procedure (write_channel) takes a vector of characters and delivers a void. Each time it is executed it writes one message into the channel (taking action as defined in procedure make_channel to deal with a full channel or a busy channel).

The second delivered procedure (read_channel) takes no parameters and delivers a vector of characters. Each time it is executed it reads one message from the channel (taking action as defined in procedure make_channel to deal with an empty channel or a busy channel).

The third delivered procedure (change_key) allows a user to lock the channel. Change_key prompts the user for a key or password. The key that is supplied will be required by procedures change_key, read_channel and write_channel before they give access to the channel. Failure to respond with the correct key will cause the access procedures to fail.

A user invokes make_channel to create a new channel. He need not know how the access procedures work. He can use them to access the channel and he can pass them to other users to enable them to access that channel. The channel users need not know the size of channel (input to procedure make_channel), nor how the procedures work. The channel is a non-local value of both read_channel and write_channel bound to these procedures when make_channel is executed. The key or password is a

non_local of all three access procedures bound to them when make_channel is executed and changed only by change_key. It does not exist outside these procedures.

Similarly, a filestore capability can be hidden inside a set of access procedures. Procedure make_read_file created from Algol68 procedure:

```
PROC make_read_file= (INT file_capability)
                        STRUCT(PROC INT read_next_int,
                               PROC BOOL read_next_bool,
                               PROC BOOL is_empty);
```

has Flex mode: (Bfile->((Void->Int) , (Void->Bool) , (Void-Bool))
where Bfile is the mode of a capability for the filestore value.

The delivered access procedures all have the file capability hidden within them. They all give access to the same file. They may be stored on backing store because all the internal values and non-locals can be stored on backing store. Different procedures may be issued to different users, providing communication through the shared data structure. No user need know the structure of the stored data, nor need they have access to every value in the structure. The access procedures may perform any desired checks before giving access to the file.

The facility to hide values in delivered procedures is not privileged but is available to any programmer. It is used to provide flexible access control, to reduce the need for privilege and to provide safeguards against misuse of procedures that operate on sensitive data.

Delivered procedures can be used where the value of a parameter is critical. To protect a user against calling a procedure with a wrong parameter, critical parameters are bound into procedures before they are delivered the user. For example, a procedure to modify a dictionary does not have the dictionary as a parameter. It is delivered to the user environment with the dictionary that it modifies bound into it. It cannot operate on another dictionary, or on any other filestore object. A user is thus prevented from accidentally or maliciously modifying a dictionary to which he has no right.

Access control

Since to use any object in Flex filestore or mainstore a user must have the appropriate capability, which controls the type of access permitted and which can be neither forged nor altered, it is clearly important that capabilities themselves be protected from theft by other users. Capabilities, like other values, are protected by hiding the only copy of the capability in the non-local values of access procedures. The protected capability can then be shared safely by issuing the capabilities for the access procedures. The protected value cannot be reached except by executing an access procedure because it does not exist outside these procedures. The capability to execute an access

procedure does not give the ability to dismember the procedure to get at the protected value. The access control provided by the capability mechanism combines with the protection provided by procedure values to give an unusually powerful and flexible form of access control that can be used, not only by the operating system, but also by any programmer, to protect values. Access procedures are ordinary procedures created by a user. No privilege is required.

Failure of procedures does not allow a programmer to gain access to protected values. An exception value is the only value delivered by a failed procedure. Values bound in to a procedure as non_locals are automatically excluded from exception values and the Flex architecture allows a programmer to protect sensitive procedures by preventing them from providing the source text capability to the exception value or by preventing them from providing any information at all to the exception value.

Before giving a user access to a protected value, the access procedure may perform whatever checks it likes to ensure that access is authorised. It may require a complicated sequence of actions. It may even record for future analysis all attempts at unauthorised access. It will not necessarily request a password. If a password is requested it need not be a single word. When an access check fails the access procedure fails, denying the user access to internal values of the procedure. Not only the protected value but also other sensitive values such as a user-defined password can be bound in as non-local values to exclude them from exception values. Having successfully executed all access checks, the access procedure gives access to the protected value, perhaps by calling the command interpreter. The user will still be executing the access procedure. The value will again be hidden from the user on exit from the access procedure.

An example of the use of access procedures on Flex is found in the protection of each user's private environment (dictionaries of name/value associations). The capabilities for the dictionaries in the environment are embedded in a procedure called a user_id procedure. Access to the environment is granted only while the user_id procedure is running and only after the protection checks (such as passwords) have been satisfied. Each user has his own user_id procedure, and he remains within that procedure until he disconnects from Flex. Invoking the user_id procedure is the Flex analogue of logging in, and exiting from the user_id procedure is the Flex analogue of logging out on a conventional system. An entire session on Flex therefore takes place during a single execution of the user_id procedure. The only names recognised by Flex before a user_id procedure has been entered are the names of the individual user_id procedures. A user_id procedure cannot be invoked from within another user_id procedure because the environment set up within a user_id procedure does not include the names/values of the other user_id procedures. Access to the passwords or to a user environment cannot be gained except while executing the

corresponding user_id procedure because the necessary capabilities are bound into the user_id procedure, and do not exist elsewhere.

Privilege
Most conventional operating systems have to allow certain privileged users to break the normal access rules to allow access to special operating system values such as those needed to archive or to use peripherals. Flex makes normally privileged values available to ordinary users by supplying them within procedures that allow them to be used only in the authorised way. The bound in values can neither be accessed nor changed by the user. For example, procedures for handling filestore, or other peripherals such as the current vdu are provided with the operating system pointers bound in as non-locals. Any Flex user can use the archive procedure, which has bound to it, hidden from the user of the procedure, the values needed to achieve the archive.

3.7 Separate compilation on Flex

The purpose of a PSE is to evolve programs. To develop programs a programmer must be able to compile source text, to test procedures, to diagnose errors, to correct incorrect parts of a procedure, to propagate changes to affected users and to keep track of changing software and of associations between different software units.

The module
The unit of separate compilation on Flex (the Flex module[14]), is fundamental to the process of program evolution. A filestore value of mode Module gives access to the compiled code resulting from a single run of the compiler, to its external specification (the data and procedure types and source text names visible to users of the compiled unit) and to the source text from which it was derived (with certain restrictions, see below). Some users arrange for a module to give access to an edfile (the log file) in which all changes to the module are recorded. A module also gives access indirectly to all the modules that it uses. The Algol68 source text of a module that uses previously compiled modules includes a capability for each used module. This has the advantage of being context independent, whereas, if the reference to used modules were by name there would need to be a context (or library) for each module to associate the module names with the correct version of a compiled unit. A module is displayed by the editor in a cartouche with a label giving the source text name of the module so it is easy when reading source text to identify the used modules.

One problem in program evolution is keeping track of which source text was used to create a particular version of a program or compiled unit. Most systems rely on the programmer to keep records or to use an appropriate set of file names to assist in this task. This can lead to confusion. On Flex this source of confusion is eliminated by binding both the compiled code and the source text from which it was derived into a single object, the module. The user can extract not only the

source text from which a module derived but also, by chaining through the used modules, any of the source texts associated with used modules (provided the text is not protected, see below). The user can always be confident that the text extracted from the module is the correct version. This relieves him of the administrative burden of recording which source text was used to create any given module. In fact the usual practice is to avoid any possibility of confusion by deleting explicit references to (capabilities for) the source text as soon as the module has been created.

If a procedure or module is to be issued to other users, the owner may choose to hide the source text by removing from the issued module the capability to access the source text. This facility is available to any programmer.

Changes

A common problem in software development is to ensure that error corrections are propagated to all users of a compiled unit. Some systems provide automatic propagation of changes in compiled units to all dependent units although the propagation does not always cover programs using the changed units. Other systems rely on manual propagation of changes aided by tools such as a program to list where a compiled unit has been used. The tools may depend on a programmer remembering to insert the necessary information in a database. The administrative burden of keeping track of changes can be reduced by such tools, but reliance on systematic use of the naming system and on separate systematic recording of all changes and all uses of a compiled unit is dangerous and error prone.

Changes to Flex modules have immediate system wide effect. A change is automatically propagated to all users of the module, to all programs (procedures) that involve the module and across all Ada libraries that contain the module. (It is possible to test a new compiled unit before actually changing the module value.) When a module is changed every instance of the module capability gives access to the new compiled unit and associated values. (The change to the module does not violate the non-overwriting filestore. Modules are always accessed through a dictionary and the user is given a new dictionary, with the old module value replaced by the new one. The user need not be aware that the dictionary has changed. It is done automatically.) Every procedure uses the most up to date version of its modules because linking the compiled code is postponed until the procedure is called from curt.

There are two ways to alter a module, a change that alters the external specification of the module and an amendment, that leaves the external specification unchanged. The amendment procedure does not permit alteration of the specification. Because different procedures are used a programmer cannot unwittingly alter an external specification. Unnecessary recompilation is avoided because modules using an amended module need not be changed. If however the external specification is

changed, each instance of the capability for the changed module is marked to show that the value must be revalidated before use. Procedures involving invalid modules cannot be loaded. Flex thus avoids the undefined results that can occur if accidentally using a procedure whose specification no longer matches the call. Capabilities for invalid module values are easily identified, and procedures exist to search for them and revalidate them, automatically recompiling only as necessary and calling the editor to invite change where non-matching interfaces are involved. This enables everything to load correctly again.

Automatic propagation of changes must not be undertaken lightly, since the results could be disastrous if a change had not first been adequately tested. On Flex changes can be tested before automatic propagation occurs.

Procedures are derived either from a module or directly from the compiled code without affecting the module. The derived procedure can be tested by calling directly from curt. It is unnecessary to form a program, create a test harness or a file of test data to provide the input parameters even when the Flex mode of the parameters is complex. The user makes up an object of the correct input mode using procedures provided for creating Flex modes and assembling complex objects. The results can be checked using procedure "show" which displays any value in a readable form. Special display procedures are not needed. This makes quick testing of individual procedures very much easier than in many systems.

Alternatively selected modules in a procedure can be replaced by the compiled units under test using a procedure called mapped_fn. The replacement has no effect on the old version of the procedure and it is unnecessary to alter any source text to achieve the replacement. Module replacements can be made only if the replacement has the same external specification as the replaced module. Replacement of modules with altered external specification is achieved by replacing the using module, altered as necessary to match the altered used module specification. This is a welcome advance on systems in which it is necessary to rename a temporary version of a procedure, to change source text in a calling procedure, to invoke the temporary version and to keep track of what has been changed, where and why.

Modules are protected from unauthorised change. Only the creator of a module will normally possess the capability to change it and attempts to change a module without the appropriate capability fail. The creator can still modify an issued module, propagating the change, because he holds the necessary change capability. The module capability gives access to the same content (i.e. the compiled code and the external specification, and, if not hidden, the source text) whether issued to another user or not.

It is possible to arrange that all changes to modules are automatically

recorded within a log file accessed through the module. The procedures to change the module record the change in the log file inviting the user to add a comment explaining the change. Other relevant information, such as the name and address of the author and even source text of the superseded version may also be included.

3.8 Relationships between filestore objects

General relationships

In most computer systems explicit relationships between different objects in the environment are provided only in databases. Usually these, if available, are provided in addition to the PSE rather than as an integral part of it. Several systems now support tree structured dictionaries, so that related objects can be held in the same sub-dictionary but it is less usual to support more complex structures, with the same object appearing in more than one place in a tree-like structure, as in the Flex edfile.

Flex does not currently support a traditional database management system. However,edfiles are database-like objects that hold typed values of any mode and implicitly express relationships by holding related values either adjacent or at different levels in the same edfile. For example, relating a set of text files such as a requirement specification, a design document, user guides etc. to the modules and procedures that implement them can be done by holding all the capabilities in the same edfile, together with the implementation, as illustrated in the edfile example, ed_date_doc. Meaning can be documented fully in the surrounding text rather than relying on the somewhat inadequate shorthand of appropriately chosen names. Multiple relationships may be expressed by holding a capability in more than one place.

The usual way to find a value on Flex filestore is either to scan a structured edfile (provided the user knows roughly where in the structure to look) or to name the value. Because of the acyclic nature of Flex filestore there is no explicit route from a value to the values that contain it. Some searches, such as trying to discover all users who possess a specific capability, may be impossible without searching the entire filestore (which may not be permitted!). It is however easy to confirm whether a container (such as an edfile, a dictionary or a module) holds the value or to search the container for a value with some required characteristic. For example, a procedure could find all objects in an edfile of specified mode or a procedure could search for a module that uses a given module. It can be useful to discover which module keeps a specific item, such as an Algol68 mode, in its external specification and hence to look at the item in more detail. Using existing procedures from the common dictionary, it was easy to write a procedure which, given a source text name, searched a container and delivered to the user a module keeping the named item in its external specification.

For reasons associated with the integrity and security of the Flex system, a user cannot deliver from a filed procedure the module from which it was derived. He can however discover from which (if any) of a set of modules to which he already has access, the filed procedure was derived.

Special relationships

There are currently some explicit relationships on Flex. There is the binding relationship between any value and its mode and a relationship binding an edfile (including source text accessed from a module) to its creation date. Flex dictionaries express the relationship between a named value, its information edfile (which may be structured) and the date of the association between name and value. The module expresses the relationships between compiled code, its source text, its external specification and its log file. The module also expresses the relationship between a module and the modules that it uses. Other explicit relationships could be added to Flex using a data structure similar to mode Module in which related capabilities would be accessible through the same object.

Versions

Versioning is a partial ordering, in which any object is superseded by its successor, but parallel versions may also exist that do not affect the currency of other versions.

Parallel versions of a procedure differing from each other in only a few modules are easy to create and to maintain on Flex. The mapped_fn procedure (which replaces existing modules in a procedure by new compiled units) can be used to create the parallel version with the differing modules replaced. (For example, a cross compiler could be a native compiler with the back end mapped to an alternative version.) The parallel version can be retained in filestore in the usual way, by naming or by storing in an edfile. The use of mapped_fn to produce parallel versions of filed procedures ensures that only one version exists of any code that has not been replaced. Corrections to modules that have not been replaced will therefore be propagated to both versions in the usual way. The propagation of changes to the replacing compilations would not be as neat as the propagation of changes to modules, and a balance probably needs to be struck to decide when a new module would be more appropriate than a replacement. If, however, a replacement compiled unit is used in only one modified procedure, as usually happens, then the inability to propagate changes automatically to other units is irrelevant.

Conventionally a sequence of versions of an item share a common name, optionally qualified by a version identifier. Flex does not have this concept. A Flex name in a given dictionary references a unique value, not a sequence of values. If a user wishes to retain a sequence of values he will probably keep them all in a single edfile, with suitable text or labels to indicate status.

Values previously associated with a name or held in a previously named value may be retrieved through old dictionaries, which are automatically retained on Flex until the next disc garbage collection. The Flex computers do not currently have infinite filestore (e.g. suitable tape drives). If they had there would be no reason why all previous versions of dictionaries could not be retained. However, with limited filestore available it has been arranged that old dictionaries can automatically be discarded by disc garbage collection. (Disc garbage collection is not automatic. It must be initiated explicitly and the user always knows when it occurs.) If another user possesses the capability for an old value it will still exist after disc garbage collection. A discarded value continues to exist as long as a capability for it is retained somewhere.

The user of a conventional system cannot usually recover a prematurely discarded file. He may retrieve an earlier version by use of the system back-up facilities, but the earlier version (if accessible) may not be the same as the discarded file and may be inconsistent with other files or dictionaries in the current filestore. On Flex a discarded value can be recovered by access to the old dictionaries, in the knowledge that the value recovered is precisely the value that was lost and not some arbitrarily selected earlier version. The fact that root pointers are the only values on Flex that can be over-written means that each root pointer gives access (indirectly) to a completely consistent snapshot of filestore.

A user worried about losing access to old versions at garbage collection can arrange to keep old versions automatically. I felt very uneasy about losing old versions of a changed module at disc garbage collection so I modified my procedures that change modules to give the option of retaining the superseded version in the log file associated with the module. This was a very easy task because I could re_use existing procedures to extract text from a module and to change modules. Other tools to handle old versions could be developed.

3.9 Some results of Flex characteristics

Separation of concerns

As illustrate in the edfile example, Curt commands are unusual in structure. The structure combined with the ability of Curt to handle arbitrarily complex data structures encourages separation of concerns. Each procedure is applied to the result delivered by its predecessor. Procedures are executable values no matter what non-local values or parameter structures are involved. It is unnecessary to combine different procedures into a single program either to enclose the non-local values or to pass results from one procedure to the next. Complex functions are achieved by applying single purpose procedures in sequence, each performing one simple function.

For example, the editor processes only edfiles. If a user wishes to

change the text of a module he first applies a procedure to the module delivering the text as an edfile and then applies the editor to the result. Similarly, he may apply a procedure to the module delivering the external specification as an edfile and then apply the editor to the result in order to display the specification. It is unnecessary to merge the distinct functions into a single tool. The separation of concerns into distinct procedures makes it easy, not only to test procedures, but also to re-use them.

Orthogonal interface and re-use

The Flex PSE is the first system I have used where there is no artificial distinction between system procedures, utilities and user procedures. The utilities are not privileged and can be re-used anywhere in the system. They are distinguished only by the fact that they are accessible to all users. Most procedures use the system supplied procedures such as the editor and the command interpreter to interface with the user making for a much more orthogonal user interface than is usual. For example the "diagnose" procedure calls the editor to display diagnostic information. The user therefore interacts with "diagnose" exactly as he interacts with the editor. He can examine on-line documentation while diagnosing a procedure error or he can obey commands to create new values and Flex modes which he may retain within an edfile if he wishes.

One of the more tiresome characteristics of some operating systems is the cumbersome way of moving from one program to another. Because programs can be invoked only from the command interpreter a user is usually required to perform the apparently unnecessary step either of returning to the outermost (operating system) level in order to invoke another program or of spawning a new process running a separate invocation of the command interpreter from which he can call other programs or utilities. Flex avoids this by allowing any procedure (including a utility) to be called in any context.

For example, if a compiler on Flex successfully compiles some source text it delivers a compiled unit to the user. If, however, errors are detected in the course of the compilation, the compiler calls the editor on the faulty source text, with marks in the text at each detected error. The editor can position the cursor at each error in turn, allowing the user to edit the source text, if necessary consulting on-line documentation or even compiling and running a separate procedure, while still in the editor. On exit from the editor, the edited text is returned to the compiler which delivers it to the caller of the compiler (usually the command interpreter). The user can then immediately invoke the compiler again on the modified edfile if he so wishes. This greatly eases the process of obtaining a clean compilation, since there is no need to exit from the compiler and to invoke the editor in order to correct errors, nor is there a need to note down line numbers and have paper listings always available.

The facility to call any procedure directly from any other is more powerful than the facility to spawn a new process to run a utility such as the command interpreter because there is no artificial limitation on the data structures passed between caller and called. When a new process is spawned most systems limit communication between spawning and spawned process to simple data such as an integer or a file name. There is also sometimes a system limit on the number of processes that can be spawned from a single parent.

The ability to re-use utilities from within user procedures is initially disconcerting, because it can be difficult to keep track of where you are. I was used to returning to the command interpreter on exit from a utility, and now I might return to the editor, to one of my own procedures or to another utility, depending only on which procedure called the utility. However the improvement in the user interface achieved by the re-use of procedures needs to be experienced to be appreciated.

Parallel processes

Flex provides facilities for parallel processing. Any procedure can launch any other procedure, to run in parallel with it as a separate process. The facility is not privileged. The Flex editor allows users to launch independent processes in parallel in separate screen windows. The command is similar to that for a normal procedure call. It allows a user to continue with foreground work in one window while the launched process runs in another.

Delivered procedures enable Flex to provide semaphores to control shared data. A Flex semaphore is a procedure shared by the communicating processes, of mode Bool -> Void . It reserves or releases a semaphore according to the boolean parameter. A call to reserve an already reserved semaphore causes the caller to be suspended until the semaphore is released by another process. Each call of procedure make_semaphore delivers a semaphore procedure with the value of the semaphore hidden inside it.

The capability to fail a launched process is a procedure delivered when the process is launched. It cannot be used to fail any other process because it is delivered with the identity of the process that it can fail bound into it. Both the launching and the launched process can provide this fail capability to another process giving the receiving process the power to fail only that process and no other.

4. Performance of Flex as a PSE

How well does Flex measure up to the listed objectives of a PSE (going through the objectives in the order given in section 2)?

1. Flex is trustworthy. The capability architecture and the non-overwriting filestore provide a very robust system in which

hardware or system failure cannot result in inconsistent filestore. Neither filestore addresses nor mainstore addresses have any meaning on Flex so they cannot be misused. All addressing is handled by capabilities which cannot be manipulated or forged. Corruption cannot be caused either by simultaneous updating of filestore objects or by partial updating. Programmers need not therefore provide software to guard against possible corruption of filestore objects.

Parallel access to a filestore object cannot cause corruption although it can result in loss of an update. If two users, working in parallel (perhaps using different copies of the capability for the filestore object) both attempt to change the same object in the same dictionary (by naming an object or by changing a module) both users will succeed but the last update to be made will supersede the first. Although such loss is frustrating, since the changes made by the first user will apparently have been ignored, the integrity of the filestore is maintained. In practice it is unlikely that two users will want to work simultaneously on the same named value in the same dictionary although if users felt strongly about this, software semaphores could be provided to protect filestore values from parallel access.

2. The access controls on Flex are unusually flexible and powerful. A value cannot be accessed without the appropriate capability which can be neither modified nor forged and which enforces correct use. Flexibility of access control is achieved by allowing programmers to hide capabilities in access procedures which they write themselves, with user defined tests to protect the values. Procedures can only be executed and not dismantled, so the internal values cannot be reached except when executing the access procedure. Failure of procedures does not enable unauthorised access to the protected values.

Read access to the source text of shared modules can be prevented by removing from the issued module the capability to examine the source text. Use of a shared module does not give the user the capability to change it. The change capability is normally retained only by the module creator.

3. The need for privilege on Flex is minimal. It is unnecessary to break the normal access rules to allow operations on privileged values because the privileged values are made available only within procedures that use them only in the authorised way. Access to special values needed to archive or to use peripherals can safely be granted to any Flex user by providing the value bound into a procedure. Procedures to communicate with a vdu and keyboard have the pointer to the specific vdu bound into them.

Code sharing between procedures does not require privilege. Code is automatically shared by procedures using the same module. Any procedure value can be called from any other so privilege is not needed to make a system procedure or utility callable from another procedure.

4. Flex gives good support for program evolution. Programmers need never be confused as to the version of the source text from which a compiled unit or procedure was derived. Source text is kept in a module. Relationships between the different modules that constitute a product are maintained by keeping the module capabilities within the source text of a using module. A log file, recording changes and possibly keeping old versions, is maintained as part of the module. Documentation and descriptive text are kept with the modules, procedures and other values in a structured edfile instead of in separate files.

A programmer cannot accidentally change the wrong module when updating a module without specification change. The requirement explicitly to request a specification change prevents this. Support for change control includes automatic propagation of module changes not only to all using modules but also across all Ada libraries and to all using programs. Tools are provided to assist in recompilation of dependent code where necessary.

Procedures are tested prior to propagation of updates without any need for temporary changes to source text. Programmers can create and call procedures directly from the command interpreter without affecting a module or writing a test stub, or they can test compiled code by selective replacement of the modules in a procedure. The diagnostic tool displays relevant source text to programmers and shows internal values in readable format. Users can examine documentation or invoke any other procedures and utilities while still diagnosing an error. This gives very powerful facilities for locating program errors.

5. Flex provides excellent facilities for using data structures to suit the problem. Modes provide explicit description of data structures and new modes can easily be defined. Values are always associated with their mode even on filestore so the structure information is not lost. The structure used within a procedure is retained when values are delivered from a procedure. Appropriate use of the structured data is enforced by the command interpreter, using procedure modes.

6. Flex is growing all the time. Adding new tools to Flex is simply a matter of writing new procedures and adding them to the common environment if they are to be shared. Augmenting Flex with new procedures or new Flex modes has no unwanted side-effects. It does not impose changes on existing procedures and modes.

7. Flex minimises the need for new software in several ways. The Flex architecture means that neither operating system nor application software need concern itself with mainstore allocation and release, filestore allocation, checks against misuse of store addresses, garbage collection, filestore inconsistency (detection or recovery) or "type of use" access controls. Mode checking by the command interpreter before procedures are called reduces the need for parameter checking within

individual procedures. The use of the Moded mode, combining a value with its mode, means that a single procedure can be provided to handle any mode without losing the mode information, so different procedures are not needed for different modes.

Re-use of modules and of procedures is fundamental to Flex[12]. Tools and utilities are normal procedures that are easily re-used within user procedures. Flex tools tend to perform simple functions because it is so easy to pass the result of one tool invocation directly to the next tool. They are therefore appropriate for re-use. No privilege is required to re-use modules from the operating system procedures. Any module may be re-used and any can be made available through the common dictionary. Although parallel processes can be spawned on Flex it is not necessary to spawn a new process to invoke the command interpreter from an ordinary procedure.

It is unnecessary for programmers to provide special software to handle exceptions. Exceptions are handled in a uniform way. The exception values provided when a procedure fails can be handled by user procedures and provide information to the diagnostic program without special software for each possible exceptional case.

8. Flex provides a very comfortable and consistent user interface. The simple command language, with its facilities for calling a sequence of procedures easily encourages simplicity in the tools. The high orthogonality results in part from re-use of the editor and the command interpreter within user supplied procedures. Because it is easy to move between the editor or the command interpreter and other procedures, Flex avoids the clumsiness of needing to re-enter the operating system between each call of a utility.

9. Flex protects the user from himself in several ways.

The risk of passing an incorrect data structure between procedures is reduced by the refusal of the Flex command interpreter to call a procedure with parameters of the wrong Flex mode.

A user cannot accidentally change the specification of a module, nor can he modify someone elses module.

Critical values can be bound into procedures to prevent the use of wrong parameters where the correct use of parameters is crucial. For example, system procedures which can affect a user environment by altering a dictionary, are available only within that environment, and have the user dictionary bound into them.

The use of values in preference to names for most Flex objects reduces the risk that a user will unknowingly access a wrong value. A name can change meaning (be associated with different values at different times or in different contexts) but a value is context independent.

Users are protected to some extent from premature deletion of filestore objects by the persistence of the object as long as a capability for it exists and by the retention (until disc garbage collection) of old root pointers. A user can recover from his error even if he has no other copy of the capability, by recovering a discarded capability via an old dictionary.

If a user issues a command which he subsequently regrets he can easily revert to the previous state by undoing the command.

10. Help is always easily accessed on Flex by calling the editor to examine the tutorial, the structured documentation file or the information provided on named values. Flex documentation is reasonably well structured, bearing in mind that no two people will ever structure a document in exactly the same way.

Machine held documentation is easier to maintain with a short update cycle than paper based documentation. The potential for inconsistency when repeating the same information in several places is reduced by inserting a capability for the repeated information in every document that refers to it. A procedure that lists all objects in a dictionary that do not have an info file assists in ensuring that information is provided when new named values are added to a dictionary.

11. Although a general database management system with explicit relationships between different types of object in the environment does not exist on Flex, there are some database like aspects to the system. Every value has an associated mode indicating the operations that are valid on it. Relationships are implied by keeping related values together in a structured edfile, repeating values in different contexts to imply multiple relationships. Such relationships are random in the sense that no rules exist which either enforce, forbid or in any way control them. There are some explicit relationships (those provided between a named value and its information file, and those provided by the module, binding compiled code to its source text, its external specification and its log file and a module to the modules that it uses). Other explicit relationships could be developed using the same sort of structure as a module.

Searches of a container (such as a dictionary, an edfile or a module) to find a specific value or a value with with some specific characteristic is easy. The structures provided by Flex edfiles must be acyclic. This is a consequence of non-overwriting, the benefits of which probably outweigh the need for indirection in creation of cyclic entities. However, it does mean that there is no way, short of scanning the entire filestore, of discovering where a given value is held. This is not as serious as it might be because propagation of corrections to modules taken from the shared dictionary does not require knowledge of who is using the modules and any container can be searched.

From the above list it is clear that Flex satisfies most of the requirements for a PSE. It does involve a total change in the way of working, which is initially difficult. Some tools that I would like to see have not yet been developed, (in particular, tools that are not interactive), but at present I can see no insuperable obstacle to their development. The toolset is constantly growing, and existing tools are being improved.

5. Summary

Flex has several unusual features that contribute to its power as a PSE and that make it different from a conventional PSE.

All access to data is hardware or microcode controlled using the capability architecture and the storage management system. The non-overwriting filestore ensures that successive values of a root pointer each give access to a completely consistent set of unchangeable filestore values. The programmer need not therefore protect his data against potential corruption, nor need he concern himself with store management. The protection provided by the hardware does not impose the performance penalties that would apply if the same functions were performed by software. Despite the non-overwriting filestore, Flex does not consume filestore particularly rapidly. Several characteristics contribute to this. The fact that only one copy of any value is held on the filestore no matter how many capabilities exist for it combine with structured edfiles to reduce the quantity of new values created when writing a new edfile. Filed procedures and modules using the same module actually share the same copy of the module instead of needing their own copy bound into an executable image.

The Flex mode system supported by the command interpreter gives the same benefits at the operating system level as is gained by data typing in programming languages. It provides visibility of data structures and confidence in the correct use of data. The programmer can create and select Flex modes freely to suit his problem and can rest assured that his procedures will not receive invalid data structures. He need not include software within each procedure to check the validity of the input.

An editor that handles values of any mode as well as text leads to the tree-like structured edfiles. These structures make it easy to hold related objects together, to document values in the edfile that holds them and to search the structures for unnamed values. The ability to call commands from an editor that handles values makes it easy to construct commands and to create and keep new values. At every interaction of every procedure it is possible to use the results obtained so far to determine future actions, without having to program alternatives in advance. The use of values as opposed to names gives a freedom which is difficult to appreciate without experiencing it. The programmer no longer has to invent names for every object he handles

nor to keep systematic records of what the names mean in different contexts.

A procedure is a value, independent of context. This has far reaching consequences. The distinction between utilities, programs and procedures disappears, so that programmers can move freely between them. Any procedure, including system procedures can be called in any context, without privilege and without spawning a new process. The amount of new software needed in an application is reduced by re-use of system and other procedures. Re-use of the editor and the command interpreter result in a consistent user interface. Procedures can be delivered from other procedures and are used to provide very flexible control of access to values, to bind critical values such as dictionaries to user procedure and to provide parallel processing.

Binding source text to the module removes the potential confusion of working with the wrong version of source text and enables the diagnostic procedure to deliver the actual text in which an error has been detected. Listings are rarely used.

Amendments to modules have system wide effect. Programmers need not therefore worry about keeping accurate records of the recipients of modules. Controls are provided to prevent accidental specification changes, use of modules with non-matching interfaces and a change to a module other than by its creator. Automatic means are provided to locate capabilities for modules with changed external specification and to recompile only as necessary. A potential change to a module can be tested without propagation of the change either by creating a procedure from the compiled unit under test and calling it directly from curt or by replacing a module in a procedure with the compiled unit under test. Replacement of modules is also used to produce parallel slightly different versions of a procedure.

Exception values, containing information about the state of a failed procedure, give great flexibility in the handling of failures. They can be handled internally within user programs or passed out through the calling procedures to the command interpreter. One result is the power of the diagnostic procedure, with its ability to present the current state of a failed program. Another is a reduction in the amount of software needed in application programs that cater explicitly for exceptions. The fact that specific values can be withheld from the exception value enhances the access control provided by procedure values.

6. Conclusions

Flex performs well as a programming support environment. It is particularly suited for developing software prototypes and for testing new ideas on software methods and designs. Once a user has learned the totally new way of working that Flex requires the system is easy to use.

The resultant improvement in software productivity has not been measured, but the absence of many of the problems that beset programmers on conventional systems must result in improved productivity.

Future work on Flex is aimed at making the PSE more widely available, and at improving the facilities. The body of software available to the user of the PSE is constantly growing. Although Flex is not yet a project support environment because it lacks the tools needed for some phases of the software life cycle, such tools could be developed.

RSRE are anxious to get reactions to Flex from a wider user community. To that end a Flex evaluation kit, running on an ICL Perq2[15], is available to users from industry, the universities and the UK Ministry of Defence. Users are supplied with the microcode and software to create a Flex PSE on their ICL Perq2, together with the machine held tutorial and documentation. It is hoped that feedback from this exercise will assist in improving Flex, with a view to eventual commercial exploitation.

Development has started on a network of Flex computers. It is possible to attach Flex to a network and to transfer data between Flex systems and between Flex and other computers. Work is currently underway on the problems of transferring Flex capabilities between Flex systems.

Implementation of Flex on existing computer systems (without re-microcoding) is being considered. Even if Flex is not transferred to existing computers as a whole, some of the ideas demonstrated on Flex could usefully be adopted on other systems. For example, the support for data typing by the operating system, provision of an editor that handles values as well as text, thus supporting a structured file system like edfiles, provision of non-overwriting filestore to improve system integrity, binding compiled code to the source text from which it derived and to a log file recording all changes to the text, automatic propagation of change to compiled units throughout the system and the ability to re-use all or part of operating system procedures in user programs.

Research has started on an on-the-fly garbage collector to avoid the automatic invocation of mainstore garbage collection which could limit the usefulness of Flex for time-critical applications.

7. Acknowledgments

I would like to thank my colleagues in Computing Division RSRE for their helpfulness and patience while I was learning to use Flex and for their help in reviewing this paper.

8. References

1. "PerqFlex Firmware" by I.F.Currie, P.W.Edwards and J.M Foster. RSRE Report 85015 December 1985
2. "Flex: A working computer with an architecture based on procedure values." by I.F.Currie, P.W.Edwards and J.M Foster. RSRE Memorandum 3500. 1982.
3. "Kernel and system procedures in Flex" by I.F.Currie, P.W.Edwards and J.M Foster. RSRE Memorandum 3626. 1983.
4. "Curt: The command interpreter for Flex" by I.F.Currie and J.M.Foster. RSRE Memorandum 3522. 1983.
5. "A software development system supported by a database of structures and operations" by Kanasaki, Yamaguchi and Kuni. Proc. of IEEE COMPSAC 1982.
6. "Software Engineering Environments" edited by H.Hunke, North Holland Publishing Co. 1980.
7. "The mechanical evaluation of expressions" by P.J.Landin Computer Journal Vol 6, No 4, pp308-320. Jan 1964.
8. "In praise of procedures" I.F.Currie RSRE Memorandum 1982
9. "Using true procedure values in a programming support environment" M.Stanley. RSRE Memorandum 3916, 1986
10. "The use of values without names in a programming support environment" M.Stanley. RSRE Memorandum 3901 1985
11. "Extending data typing beyond the bounds of programming languages" M.Stanley. RSRE Memorandum 3878. 1985.
12. "Some practical aspects of software re-use" M.Stanley and S.J.Goodenough. in "Software Engineering Environments" edited by I.Sommerville, Peter Peregrinus Ltd, 1986.
13. "Integrity and the Flex Programming Support Environment" M.Stanley. RSRE Memorandum 3915 1986.
14. "Some IPSE aspects of the Flex project" I.F.Currie in "Integrated project support environments" edited by J.McDermid, Peter Peregrinus Ltd 1985.
15. "Flex and the ICL Perq2" M.Stanley. University Computing Vol 8, No 2, Summer 1986.

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	3. Agency Reference	4. Report Security Classification
	Report 86003		UNCLASSIFIED

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
AN EVALUATION OF THE FLEX PROGRAMMING SUPPORT ENVIRONMENT

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.
STANLEY, M			Aug '86	

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

15. Distribution statement

UNLIMITED

Descriptors (or keywords)

continue on separate piece of paper

Abstract

This paper gives a personal reaction to the Flex Programming support environment, developed at RSRE, Malvern. It discusses the new approach to program development necessary to take advantage of the environment, and evaluates the environment against some criteria for programming environments in general.


FILMED
6-8