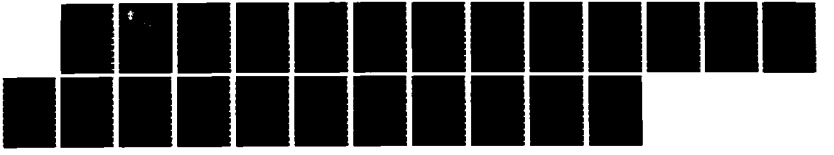
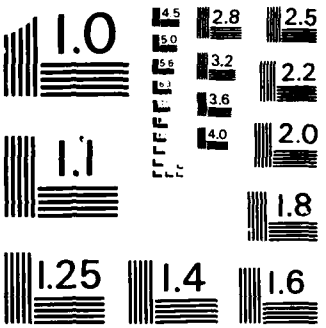
AD-A169 875 USING TRUE PROCEDURE VALUES IN A PROGRAMMING SUPPORT ENVIRONMENT(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) M STANLEY FEB 86 RSRE-MEMO-3916 1/1

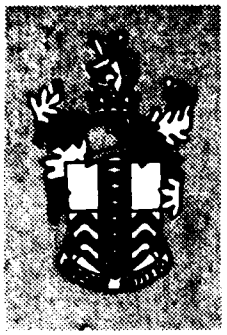
UNCLASSIFIED DRIC-BR-99598 F/G 9/2 NL





UNLIMITED

BR99598



RSRE
MEMORANDUM No. 3916

ROYAL SIGNALS & RADAR ESTABLISHMENT

USING TRUE PROCEDURE VALUES IN A PROGRAMMING SUPPORT ENVIRONMENT

Author: M Stanley

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

UNLIMITED

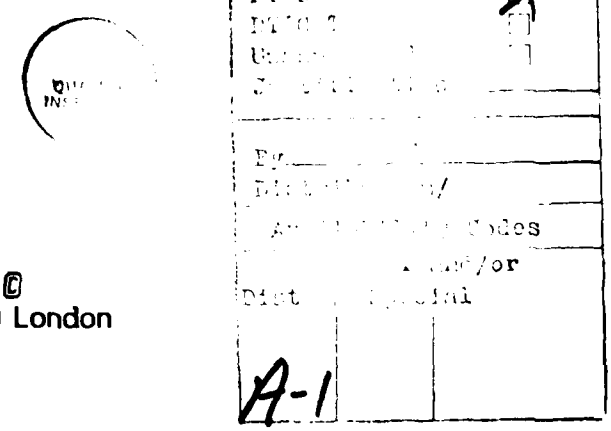
Royal Signals and Radar Establishment

Memorandum 3916

AUTHOR: Margaret Stanley.
DATE: February 1986

Using true procedure values in a programming support environment

Procedure values, as provided in the Flex programming support environment (PSE) developed at RSRE, Malvern, are discussed. A distinction is made between a context independent procedure value and a context dependent procedure. The context independent procedure is shown to provide some useful facilities to the programmer and to the developer of a PSE.



Using true procedure values in a programming support environment

CONTENTS

Using true procedure values in a programming support environment

1. Introduction

In most conventional computer systems a procedure is not a context independent value. The procedure code needs a context (its non-locals) to make it executable. The consequences of using a Programming Support Environment in which every procedure is a context independent value, bound to its context when the procedure is declared, are far reaching. The use that can be made of procedure values is discussed, contrasting the effect of using procedure values with more conventional means of achieving the same ends.

2. FLEX

The Flex Programming Support Environment (PSE) treats procedures as true values in the sense of Landin [5,6]. The PSE, built on the Flex capability architecture[1,2] developed at RSRE, Malvern, supports the efficient use of context independent procedure values. Flex is an interactive PSE that is noticeably different from other PSEs. Designed to simplify the development and maintenance of complex software, the PSE development since the first Flex architecture came into use in 1978 has been mainly a response to requests from programmers using the system. Procedure values are fundamental to the Flex PSE. They work with the capability architecture to provide a system of very high integrity. With a command interpreter that can handle values of any structure[3,4] they give a very flexible PSE. It has a large software base including all normal operating system facilities and many other procedures, including compilers for Algol68 and Pascal. An Ada(*) compiler is near completion and an ML compiler is under development.

The Flex capability computer architecture has (so far) been implemented in microcode on four hardware configurations, the most recent being the ICL Perq. The implementation with which I am most familiar is a multi-user system in which 3 Flex computers share a common filestore and common peripherals.

A full description of Flex is beyond the scope of this paper, which will discuss procedure values and their use, as demonstrated in the Flex PSE.

*Ada is a registered trademark of the US DoD.

3. Procedure values

3.1 What is a procedure value?

A procedure value is an executable value that consists of the procedure

code, constants and a set of non-local values. A local workspace is supplied when the procedure is called. The non-local values are those values (including any values declared in a separately compiled unit) that are used in a procedure but declared outside it. Procedures on conventional machines are represented not by a value but by the address of the procedure code. The assumption is made that the procedure can find its non-locals during execution using suitable pointers on the stack frame. On Flex the non-local values are preserved by the existence of the procedure value, whereas on most conventional systems the non-local values are not bound to a procedure, they are associated with a procedure only when it is called. The run-time system must then search the various stack frames to find the correct non-locals to associate with the procedure. A procedure can be executed only while its non-local values remain in existence.

For example, consider a procedure defined by:

```
DECS proc_m:
  INT counter:=0;
  PROC proc=(INT input_parameter)BOOL:
  BEGIN
     .......
     counter:=counter+1;
     ........
     counter< input_parameter
  END
KEEP proc
FINISH
```

Counter is a non-local value of proc which is incremented every time proc is executed. When proc is executed on a conventional system, perhaps from within an environment totally different to the environment in which it was declared (i.e. with a different set of values in the enclosing scope), the run-time system must find the environment in which proc was declared to find the correct value of counter.

On Flex every procedure is a context independent, executable value. It is created in mainstore (with code, constants and non-local values bound together) when the procedure is elaborated (i.e. when the declaration is encountered). In the example given above, proc exists as an executable value independent of any enclosing environment, with its non-local value, counter, bound to it as soon as it is declared. The mechanism needed to find the non-local values when a procedure is called are therefore much simpler than on conventional systems. A compilation system that includes facilities for mainstore garbage collection does not need to set up a complex mechanism to preserve non-local values that are bound into procedure values.

The procedure value can be treated like any other value in that it can be

passed to other users, delivered from other procedures and used as a parameter to other procedures without taking special steps to ensure that the non-local values can be found.

3.2 Executing a procedure value

A procedure value can be called from within a user procedure or, because the value is context independent, it can be executed directly by calling it from a command interpreter. The effect is exactly the same.

On conventional systems a procedure cannot be called directly from the command interpreter because it is not a value. It needs a context. Certain procedures, (which I shall call main programs) can be converted into something called a program which is a context independent value that can be called from the command interpreter. Main programs usually have to conform to certain rules (imposed by the programming language, the compiler and the separate compilation system) to enable the system to convert them to a context independent value.

For example:

```
DECS prog_m
USE proc_m:
  PROC prog=VOID:
  BEGIN
     .......
     WHILE proc(5) DO .....   OD;
     .......
  END
KEEP prog
FINISH
```

On conventional systems proc is not a value that can be executed without an environment, as provided here by prog. Procedure prog is written using proc-m and all its declarations as a main program that can be converted to an executable program that can be called from a command interpreter. On a system that has procedure values prog is unnecessary. The procedure value proc has counter bound into it and so can be called directly from the command interpreter.

On conventional systems main programs often differ from other procedures in that main programs cannot usually be called from other procedures. To be executed a main program must be converted into a program and called from the command interpreter. On a system with procedure values any procedure can call any other. Procedures need not be partitioned into two groups, those that are main programs and those that are not main programs.

Those conventional systems that do allow a main program to be called from a procedure usually implement the call not through the normal procedure call mechanism but by spawning a separate process in which the called program will run in parallel with the calling procedure. Although procedures called using the normal mechanism can accept arbitrarily complex parameter values and return arbitrarly complex values to the caller as well as sharing non-local values with the caller, procedures that are called by spawning a new process can usually communicate only through specially programmed interfaces, or through values written to filestore or handled by the command interpreter. Communication is therefore less convenient than with the normal procedure call.

A procedure value is an executable value in mainstore but values that are to be retained long term need to be written to backing store. Although a procedure value is not normally held on backing store, on Flex an analogue of a procedure value, called a filed procedure, can be retained on backing store. A mainstore procedure value is created from its filed analogue when it needs to be loaded for execution. This is done automatically when the filed procedure is called from the Flex command interpreter. The Flex user need make no distinction between calling a mainstore procedure value and calling a filed procedure value.

The filestore image of an executable program on a conventional machine is called an executable image. An executable image is usually a filestore value with its own copy of the code of all its procedures and its own copy of all non-local values of those procedures, with all links resolved and with a single entry point. Each program loaded from an executable image normally contains its own copy of all its procedures except for a few privileged procedures which can be shared between programs. Although normal procedure calls will not usually involve loading new copies of procedures already in use, if a procedure calls a main program by spawning a new process the spawned program may use new copies of all but a few privileged system procedures. With procedure values there is no need for executable images. A Flex filed procedure is not the same thing as an executable image. It does not contain copies of all the separately compiled units, it contains references to the required separately compiled units. The procedure values of all used procedures are created when the declarations are elaborated during loading. Since the Flex filed procedure does not contain copies of all its constituent procedures, the loading process can take advantage of this to re-use procedure values that are already in mainstore, thus enabling sharing of common procedures at run-time. Thus procedure values help to reduce the number of copies of procedure code needed by encouraging sharing of code.

3.3 Consequences of procedure values

Increased orthogonality

With executable procedure values the artificial distinction between main programs and other procedures disappears. Because Flex supports procedure values all tools or programs in the Flex PSE, whether system provided or user provided, are just procedure values. There is no distinction on Flex between a procedure value and a program or tool. The effect of calling a procedure from the Flex command interpreter, curt[3] (itself a procedure) is the same as that of calling it from any user procedure. Similarly any programmer can, without privilege, call procedures such as the system utilities from his own procedures. System utilities such as the editor and the command interpreter are procedures that can be (and are) used by other procedures to communicate with the user.

When a procedure value is called from curt the data required by the procedure are the parameters of the procedure, and the result of obeying the procedure is delivered to curt in the same way as the result is delivered to any other calling procedure at the point of call. For example proc, supplied with an integer parameter, will deliver a boolean result whether called from prog or from the command interpreter. Each call of proc will increment counter. If proc were always called with input_parameter=5 then the fifth call of proc would be the first to deliver the result 'FALSE'.

Flexible communication between programs

A command interpreter that can call any procedure directly must be capable of handling procedure parameters and results of arbitrary complexity. The Flex command interpreter does not impose any artificial restriction on the data types of the parameters or of the results of procedure calls. Arbitrarily complex values can be delivered from a procedure directly to the command interpreter for input to another procedure or for retention for future use. This is in contrast to several conventional systems in which a procedure that is a main program must have only very simple values (such as a single integer) as input parameters and can deliver only simple data structures as results. More complex values, if needed, must be supplied or delivered through values read in from or written to backing store or peripherals by the program.

Communication between procedure values called from the Flex command interpreter does not rely on one procedure writing values to filestore so that the next can access them, as is necessary for inter-program communication on many conventional systems. The parameters and the result of a procedure call may be a mainstore value of arbitrary complexity.

Adaptability

Procedure values can be called from anywhere. They behave the same whether they are called directly from the command interpreter or from another procedure. There is no need to anticipate the calling pattern and write a program to implement it, thus restricting the context in which the procedure is used. Users can re-use procedures in unanticipated ways and combine them in unanticipated ways by calling them directly from the Flex command interpreter.

Separation of concerns

One consequence of calling procedures directly from a command interpreter able to handle arbitrarily complex data structures is separation of concerns. Single purpose procedures can exist as executable entities no matter what non-local values or parameter structures are involved. They do not have to be embedded in programs. There is no need to combine different procedures into a single program to enclose the non-local values; to pass results from one procedure to the next or to perform a complex function. Complex functions are achieved by applying single purpose procedures in sequence, each performing one simple function. Each procedure is applied to the result delivered by its predecessor.

For example, consider four distinct procedures.

```
PROC line=(FILE edfile)VECTOR [] CHAR:
        (extracts character strings from an editable file)
PROC convert=(VECTOR []CHAR line)VECTOR []INT:
        (converts the character string to a vector of integers)
PROC mean=(VECTOR []INT ints)INT:
        (calculates the mean of a vector of integers).
PROC sd=(VECTOR []INT ints)INT:
        (calculates the standard deviation of a vector of integers)
```

A user wishing to discover the mean of some integers held in character form in an editable file would call the first three procedures in turn from the command interpreter, delivering the result of each call as input to the next call. If he later decided he also needed the standard deviation he would need only to write the last procedure and call it on the result delivered by the second. It is unnecessary to combine the procedures into a single program to allow them to pass the result of each call to the next procedure nor need the results be written to filestore to be passed between the procedures.

The separation of concerns applies equally to system utilities. The Flex editor processes only editable files. If a user wishes to change the text of a module (a value giving access both to compiled code and to the text from which it was derived), he first applies a procedure to the module delivering the text as an editable file and then applies the

editor to the result. Similarly, he may apply a procedure to the module delivering its external specification as an editable file 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 to re-use them.

Testing

When a procedure has been written, it can be tested directly by calling it from the command interpreter. The procedure value does not need to be put into a test harness as is required in many other operating systems. The results are delivered directly to the command interpreter and can be examined (using other procedures) without writing them to filestore. This encourages users to test procedures individually as they are written rather than waiting until they can be combined in some larger program or procedure.

3.4 Delivering a procedure value

One consequence of treating a procedure as a value, with its non-locals bound into it, is that 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 or protected from a caller of the delivered procedure.

For example, consider a procedure, make_channel, that creates a channel for passing messages and a pair of procedures for accessing that channel. (Each message consists of a vector of characters.) Make_channel takes an integer giving the size of channel (i.e. the number of messages it can hold). It delivers two procedures, write_channel that writes a message into the channel and read_channel that reads a message from 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.

The Algol68 mode of procedure make_channel is:

```
PROC make_channel=(INT size)
                    STRUCT(PROC(VECTOR[]CHAR)VOID write_channel,
                           PROC VECTOR [] CHAR read_channel):
```

The first delivered procedure (write_channel) takes a vector of characters and delivers a void. Each time it is executed it writes one message (a vector of characters) 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 user who invokes make_channel need not know how read_channel and write_channel work. He can use the procedures to access the channel and he can pass the the delivered procedures to other users to enable them to access that channel. The users of procedures read_channel and write_channel 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.

Note that this example cannot be implemented using only a stack-based machine. When the procedure make_channel returns, delivering the procedures that use the channel, one cannot destroy the channel declared locally in make_channel because it is still used in the delivered procedures. It continues to exist as long as either procedure of which it is a non-local (read-channel and write-channel) exists. Each call of make_channel creates a new channel with a new pair of procedures for using the channel.

Another way of using procedure values is to use curried functions to bind a parameter value to the delivered procedure. The delivered procedure can still take parameters. For example a procedure, p, that could be written to take two parameters, a and b:

p(a,b)

could alternatively be written as a procedure q ,taking the first parameter, a, delivering a procedure r, such that r(b) gives the same result as p(a,b)

i.e. p(a,b)=q(a)(b)

The delivered procedure q(a)=r can be used without knowledge of or access to the value of a, which is bound into it when q is executed.

3.5 Using delivered procedure values

Delivered procedure values can be used to solve a number of common problems in programming environments. The facilities provided are not privileged. They can be used by any programmer.

Protection from unauthorised access

Procedure values provide a particularly flexible and powerful way to protect values. Any value can be hidden by enclosing the only copy of it within a procedure. The protected value cannot be reached except by executing the protecting procedure because the value is bound into the procedure and does not exist elsewhere.

The protecting procedures are ordinary procedures created by a user.

No privilege is required. Before allowing a user to reach a protected value, the protecting procedure may perform whatever checks it likes. It may require a complicated sequence of actions. It may even record for future analysis all attempts at unauthorised use. It will not necessarily request a password, although it may do so. If a password is requested it need not be a single word. When a protection check fails the protecting procedure fails, denying the user access to the internal values of the procedure. Having successfully executed all protection checks, the protecting procedure allows the user to reach the protected value, perhaps by calling the command interpreter. The user will still be executing the protecting procedure. The value will again be hidden from the user on exit from the protecting procedure. The protecting procedure can be given to anyone in the knowledge that they still need to satisfy the built in checks before they can reach the protected value.

Those values in Flex which require some degree of access control to preserve system integrity are represented by capabilities. The capability mechanism provided by the Flex architecture ensures that values represented by capabilities can be used only in the way authorised by a capability and only by the holder of the appropriate capability. A capability can be created and modified only by the Flex microcode although capabilities can be treated like other values in that they can be held on filestore, used in procedures and can be passed to another user, giving the other user access to the controlled value. There are mainstore capabilities that control access to mainstore objects (such as procedures), filestore capabilities that control access to objects on filestore and remote capabilities that control access to remote facilities (e.g. objects on other Flex computers). In a sense a capability is a pointer created on behalf of the user by the microcode, but the capability also contains information on the type of use (read only; read/write; execute) that will be permitted by the microcode. Users can pass capabilities to other users.

A capability for a procedure value (also called 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. It is not possible, using software, to dismember a protecting procedure to discover the actions required to gain access to the values it protects.

Like other values, capabilities are protected from theft by hiding them inside protecting procedures. 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.

The flexible access protection provided by procedures is used on Flex to protect a 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. 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. 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.

Information hiding

A critical value (such as a password) may be needed within a procedure. If such a value is actually a non-local of the procedure rather than a data value held in some associated data area then it cannot be illegally accessed because it does not exist outside the procedure. Any value can be protected in this way, not just passwords.

It can be desirable to hide from the user of a data structure the details of the structure itself. If a structure is complicated it is often unnecessary that the user be aware of the details. He needs only to have access to or to be able to update the values contained in the structure. A hidden structure may also include values that a user does not need to know. Procedure values allow programmers to provide procedures to create and use a data structures whose implementation and detailed structure is hidden. In the make-channel example the user or users of a channel need not know the channel structure. They interact with it only through the delivered procedures.

The hidden structures may be mainstore values as in the make_channel example or they may be filestore values. For example, consider a procedure of Algol68 mode:

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

The delivered procedures all give access to the same filestore value, identified by the file_ptr parameter to make_read_file. The set of delivered procedure values may be stored on backing store because all the internal values and non-locals can be stored on backing store.

A specific data structure, whether it be a mainstore or a filestore value, may be shared by a number of users, with different operations on the values available to different users. None of the users need know the structure of the stored data, nor need they have access to every value in the structure. The caller of make_read_file can, if he wishes,

deliver the different file access procedures to different users so that one user can only discover whether the file is empty, another can read the boolean values and a third can read the integer values. Each user is interacting with the same data file, but has only limited access to the values in it.

Sometimes a programmer may wish to retain the freedom to change the internal structure of a particular kind of data value. If a value is always created by a procedure that delivers, instead of the actual value, a set of procedures for accessing the value, then the internal structure of the value can be changed without affecting users, provided the external specification of the delivered procedures for using the structure remains unchanged.

Binding critical values

It is possible using procedure values to permit direct use of critical operating system procedures without the risk that they will be abused by supplying them with wrong parameters. Programmers can embed critical values in procedures both to prevent unauthorised access and to prevent the use of wrong parameters where the correct use of parameters is crucial. This is achieved by embedding the value in a procedure delivered from another procedure which performs the necessary binding. In particular critical parameters to system procedures can be embedded in the procedures supplied to users, so they cannot be changed. The same kind of protection can be applied to any procedures.

On some conventional systems, a peripheral device may be booked for use by calling an authorisation procedure. The operating system then needs to check that the user requesting access to a device is the authorised user and is using the device that he booked and not a different device. If the device identity is supplied as a parameter to the using procedure a user could supply a wrong parameter to gain access to a device he had not booked. If the using procedure is created when the device is booked, with the identity of the device embedded in the using procedure then the user need not supply the device identity as a parameter. The created procedure is issued to the authorised user and the device remains booked until the user relinquishes the specially created procedure by leaving the procedure to which it was issued.

For example, procedures that operate on the vdu screen need the identity of the current vdu. Each user can be supplied with procedures containing the basic code for displaying on a vdu. The pointers identifying the current vdu are bound to them when the procedure values are created at run-time. The supplied procedures all use the same basic code but they will only affect the vdu to which they are bound. The user does not need to have the pointers identifying the vdu. Indeed he has no access to them and cannot replace them with other values to gain access to another vdu.

On some conventional systems procedures that handle dictionaries must be supplied with the dictionary by the user. If supplied with an incorrect dictionary parameter they may run with disastrous results. Flex procedures to access a dictionary do not have the dictionary as a parameter. They are delivered to the user with the dictionary bound into them. They can be retained on filestore because the bound-in value (the dictionary) is a filestore value. They 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 or even from reading it. He does not need to know the internal structure of the dictionary, since all access is through procedures. Different functions such as modifying the dictionary, delivering a named value from a dictionary, adding a named value to the dictionary or displaying the content of the dictionary will all be bound to the same dictionary. Another user will get a similar set of procedures bound to a different dictionary parameter. Where a dictionary is shared procedures to update it may be restricted to a single user.

Communication

On a conventional system communication between programs called from the command interpreter is usually achieved by writing the shared data to filestore for later use by another program. As already indicated, Flex procedures called from the command interpreter can deliver arbitrarily complex data structures to the command interpreter for use by other procedures, without need to use filestore.

Using procedure values Flex achieves a more flexible and safer form of communication. Private communication between procedures is achieved by sharing non-local values, as in the make_channel example. If two or more procedures are bound to the same non-local value they can use it to communicate without any interaction from any external process such as the command interpreter. The shared value cannot be accessed from other procedures. A shared non-local value can be used for communication between users or for passing data between the procedures of a single user. Procedures communicating through shared data are bound to the non-local data when the procedure values are created and the data continues to exist as long as a procedure value that is bound to them exists. In the case of procedures communicating through a filestore value, the procedures and the shared value can be retained on backing store between user sessions.

3.6 Some other uses for procedure values

Procedure values can be used in many different ways to provide facilites which might be more difficult to implement in other ways. Some of the uses are described below.

Parallel processes

Procedure values provide a neat way to implement parallel processing. A process on Flex is a chain of active procedure calls. Each process is therefore a procedure value that can be launched from any other procedure (including from the command interpreter). A procedure called make_process delivers a procedure known as the soft_interrupt procedure containing the identity of a new parallel process. Soft_interrupt takes a procedure as parameter. When soft_interrupt is called for the first time it launches a new parallel process, calling its parameter as the top of the process chain. Subsequent calls to soft_interrupt (with any parameter) fail the launched process. The launched process can be controlled only through the soft_interrupt procedure, which holds the identity of the process.

A process can explicitly give the ability to cause it to fail to another process. The process calls a procedure which delivers a procedure with the identity of the calling process bound to it. If called, the delivered procedure will cause the bound-in process to fail. This procedure can be delivered to another process thus giving the receiving process the power to fail the bound-in process.

Interprocess communication can be achieved by sharing non-locals as with any other procedure values. If an outermost procedure launches several parallel processes with several separate calls of make_process, the separate processes communicate by sharing non-local data. Control of parallel access to shared data can be achieved by using semaphores. A semaphore is a procedure shared by the communicating processes, with the value of the semaphore hidden inside the semaphore procedure (as in the make_channel example) by a make_semaphore procedure. Each semaphore procedure, created by a call of make_semaphore, is a procedure taking a boolean and delivering a void (i.e. the empty result). Each call of the semaphore procedure either reserves or releases the semaphore. A call to reserve an already reserved semaphore causes the caller to be suspended until the semaphore is released by another process.

Privilege

Procedure values can be used to provide normally privileged operations in an unprivileged way. For example, most conventional operating systems have to allow certain privileged users to break the normal access rules to allow access to operating system values. Privilege is needed to access the special values needed in order to archive or to access peripherals. Any Flex user can have access to the archive procedure, which has bound to it, hidden from the user of the procedure, the values needed to achieve the archive. As already mentioned, procedures to communicate with a vdu and keyboard have the pointer to the specific vdu bound into them.

Users do not need privilege to use the sensitive operating system capabilities on Flex because the privileged values are made available only within procedures that use them only in the authorised way.

Privilege is not needed on Flex to make a system procedure or utility callable from another procedure. Any procedure value can be called from any other. Code sharing between procedures does not require privilege because the procedure value makes the executable image unnecessary.

Re-use

It is widely recognised that the failure by the software industry to re-use existing software in new products contributes to the high cost of software development. One reason for the failure to re-use software is that existing software products are monolithic. Programmers cannot pick up only the parts that they require. They are forced to have a whole program or nothing. An additional factor is the difficulty on many conventional systems of calling a main program from within an application procedure and the risk of permitting use of privileged procedures by non-privileged users.

General re-use of procedure values in new contexts is easy because the user of a procedure can rest assured that the procedure will always behave in the same way, whether called from his own program or from the command interpreter. Its action is totally independent of the context in which it is called. Procedures are included in a user procedure by providing the separately compiled units defining these procedures for inclusion in user procedures. Since most operating system procedures can be unprivileged, even when handling privileged values, they can be made available for re-use. It is normal on Flex to re-use system procedures such as the editor and the command interpreter to interface with the user. This leads to a much more consistent user interface than is usual, as well as reducing the number of new procedures written by users to interface with the screen.

It is sometimes difficult on conventional systems to allow programmers to re-use code in a new context, particularly code that uses operating system values (normally protected from general access) as parameters. Procedure values allow the same code to be bound to the different parameter values before being delivered to users as procedure values. 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. Procedures for handling dictionaries are provided with the dictionary bound in.

Ada packages

On a system such as Flex, where procedures are values, Ada packages

can be treated as procedures delivering the procedures declared in the visible part of the Ada package. For example consider an Ada package used to define a structure such as a stack and a set of procedures for handling the structure such as push and pop. The Ada package delivers the two procedures, push and pop just as procedures are delivered in the make-channel example. Each use of package stack delivers a new stack embedded in the two new procedures.

```
package STACKINT is
  procedure PUSH (I:INTEGER);
  function POP return INTEGER
end;
```

Unfortunately, Ada imposes the limitation that if the package specification contains simply the procedures push and pop, with the stack embedded in the procedures then it can create only one stack in any compiled unit that uses it.

```
with STACKINT; use STACKINT;
.......
push(42);    --pushes 42 onto built in stack

--cannot get another stack, with
--separate push and pop procedures
--from the same package specification
```

If more than one stack is needed there are two possible solutions. One solution is to declare the stack itself as a data type within the package specification with each procedure having the stack as a parameter.

```
package STACKS is
  type STACK is limited private;

  procedure PUSH (S:STACK;I:INTEGER);
  function POP (S:STACK) return INTEGER
private ..........
end;
```

then

```
with STACKS; use STACKS
S,T:STACK;
.......
push(S,42);    --pushes 42 onto  stack S
push(T,pop (S)); -- takes value from stack S
                 -- and puts it onto stack T
.......
```

Alternatively the stack may be declared as a generic package with a new instantiation for each new stack.

```
generic
type X is private;
package GSTACKS is
  procedure PUSH (I:X);
  function POP return X
end;
```

then

```
with GSTACKS;
S is new GSTACKS(INTEGER);
T is new GSTACKS(INTEGER);
.......
S.push(42);    --push 42 onto stack S

T.push(S.pop ); --take value from stack S
                --and put it onto stack T
.......
```

Thus procedure values are useful in the implementation of Ada packages, although Ada enforces a somewhat more complicated implementation of stacks on a user than can be achieved simply by the use of procedure values.

Pictures

Procedure values are used on Flex to extend the power of the basic editor to handle diagrams as well as text. The editor on Flex can handle not only text strings but also values of many different kinds. However it does not include facilities for displaying and editing graphs or other forms of diagram within the text. In order to accommodate these needs the basic editor has been extended to handle objects called pictures[7] which provide for displaying and editing diagrams. A picture is a value manipulated by the basic editor that includes not only the data to be displayed but also a set of procedure values that tell the editor how to display, edit and store the diagram. Each different kind of picture has its own definition which includes the structure of the data and associated procedures for displaying the data, editing the data and storing the data on backing store. Each new diagram is defined by supplying the picture definition to a procedure to make a new picture of that type. A picture procedure is delivered which converts the data structure into a value handled by the normal screen editor. The delivered procedure has bound into it the procedures for displaying the data structure and editing, as supplied in the picture definition. New kinds of picture with arbitrary properties can thus be introduced. This is only possible because Flex allows the use of true procedure values and allows the procedure values to be placed on backing store.

4. Conclusions

The use of a procedure as a value independent of context has resulted in a PSE with several unusual and desirable features.

Access controls are extremely flexible and powerful when provided using procedure values. Any programmer has the right to enclose any value in a protecting procedure which can impose any tests desired before granting access. Since procedures can only be executed and not dismantled, the internal values cannot be reached except when executing the procedure.

Data structures can easily be hidden within a set of procedures that give access to the structure without revealing its form. The structure can be modified without affecting the user of the issued procedures, provided that the issued procedures retain the same external specification. Data can safely be shared between procedures without making it accessible to any who do not have copies of the procedures, thus avoiding the danger that other users may gain unauthorised access to shared data.

Procedures can be issued safely to other users even when they use critical parameters, by binding the user-dependent critical parameters, such as the user's dictionary, into procedures delivered to a user. The user is therefore protected against calling such procedures with a wrong parameter. Procedures that would normally be regarded as privileged can be issued to ordinary programmers because the privileged values can be bound into the issued procedures and thus hidden from the user. There is no need to deny the right to use the privileged values.

Programs are unnecessary. The artificial distinction between procedures that are main programs and procedures that cannot be formed into programs disappears. All procedures are equal. Any procedure can be called either from the command interpreter or from any other procedure. A highly consistent user interface is achieved because programmers can re-use operating system procedures and other software more easily than on conventional systems, without needing any special privileges. The procedure value behaves the same way in every context.

Procedure values can easily share the procedure values that they use and parallel processes are simply procedure values that are executed in parallel with other procedure values. Interprocess communication involves sharing data between the different processes and has the same flexibility and protection as for shared data between any other procedure values.

A PSE such as Flex that supports the use of procedure values thus provides some very useful facilities particularly for enforcing

integrity. 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.

5. 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. "Curt: The command interpreter for Flex" I.F.Currie and J.M.Foster. RSRE Memorandum 3522. 1983.

4. "Extending data typing beyond the bounds of programming languages" M.Stanley. RSRE Memorandum 3878. 1985.

5. "The mechanical evaluation of expressions" P.J.Landin Computer Journal Vol 6, No 4, pp308-320. Jan 1964.

6. "In praise of procedures" I.F.Currie RSRE Memorandum 3499 1982

7. "Extending the Flex graphics editor: an object oriented approach" P.W.Core and J.M.Foster RSRE Memorandum in preparation, 1985

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
	MEMORANDUM 3916		U/C

5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location
	Royal Signals and Radar Establishment

5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title

Using true procedure values in a 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				

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

15. Distribution statement

Descriptors (or keywords)

Unlimited

continue on separate piece of paper

Abstract

Procedure values, as provided in the Flex programming support environment (PSE) developed at RSRE, Malvern, are discussed. A distinction is made between a context independent procedure value and a context dependent procedure. The context independent procedure is shown to provide some useful facilities to the programmer and to the developer of a PSE.

END

DTIC

8-86