XEROX **Inter-Office Memorandum**

To	MPL group	Date	April 21, 1972
From	Butler Lampson	Location	Palo Alto
Subject	New control transfer mechanism	Organization	PARC

1. Introduction

This memo describes the current state of a new design for MPL inter-module transfers of control and frame allocation. The goals of the design:

1) Clear separation of control transfer from determination of the context in which the new module will run.

2) A single control transfer mechanism which can model all existing mechanisms (port call, procedure call, signal).

3) Ability to connect any kind of exit from a module with any kind of entry.

Other desirable properties which seem achievable within the general framework described below but need further thought:

4) Accessible facilities for specifying the context of a module in arbitrary ways.

5) Accessible facilities for specifying signal propagation paths.

6) Conventions for displaying the current control state. At the moment we only understand how to do this well for procedure calls (using a backtrace).

2. Control transfers

Control always enters a module through an <u>inport</u>, which simply consists of a <u>transfer location</u>. Since inports cannot be moved freely, additional information can surround the inport, i.e. the input can be imbedded in a record. We will call fields of this record other than the transfer location <u>parameters</u>.

When control enters through the inport, the registers are set up in a standard way and a transfer is made to the transfer location.

Control always leaves a module through an outport, which is simply a pointer to the inport through which control is to enter the new module.

The primitive for control transfer is

xfer (o, i, a)

where o is the outport, i is a pointer to the inport through which control should be returned, and a is a pointer to the arguments. In general the module doing the xfer must first construct the return inport.

Normally the transfer location of an inport is an address in the static storage of the context being entered, and there is one parameter which is an address in the module to which control should go. The location of the inport itself provides a second parameter whose interpretation depends on the type of transfer. Note, however, that the basic transfer mechanism does not know that this is the standard way of using it.

3. Describing the control state

A procedure-based language like Lisp has a very nice way of describing the control state, called a backtrace. In the present imperfect state of our understanding of inter-module control transfers, we have no chance for such a clean mechanism. Some improvements on the present chaotic situation do seem to be possible, however.

The basic entities which are related by control transfers we will call <u>contexts</u>; there is a one-to-one correspondence between contexts and inports. We will require that every context contain the following things (normally declared to the compiler, which leaves information about their location in the code segment for the context):

1) an <u>owner</u> outport. These are expected to define a tree called the owner tree. The owner outport specifies the target for control faults (i.e. the context which will get first chance at the resulting signal). It has no other function except to guide display of the control state.

2) a <u>signal path</u> outport which specifies where a rejected signal should go next. Perhaps this specification can be overridden by the catch phrase which does the rejection.

3) a list (possibly empty) of <u>key</u> outports (see the discussion of links below).

4) possibly a <u>return</u> outport. If context A has a return outport which points to context B, then B is A's <u>caller</u>. The list of contexts obtained by following the chain of return outports starting from A is the <u>return chain segment</u> based on A. If A is not anyone's caller, its return chain segment is a <u>return chain</u> and corresponds to a stack in the present implementation.

A context which does not have control may also have a <u>current</u> outport, which is the one over which it has just given up control.

If context A has an outport P which points to context B, A is connected to B by P. If B is also connected to A by Q, and both P and Q are either key or current, the P and Q form a link between A and B. This definition is intended to characterize certain familiar relationships among contexts such as the producer-consumer relation. It is not clear how well it does this.

All this apparatus gives us three ways of displaying the control state of a computation, i.e. the current relationships among the contexts.

1) Return chains can be displayed linearly; such a display is called a backtrace.

2) Backtraces may be connected in pairs by links. If the links arise from producer-consumer relations the resulting display has a pleasing two-dimensional structure.

3) Contexts without return ports can be displayed according to their position in the owner tree. This is especially convenient when the tree arises from a maze search.

4. Storage allocation

To make the new control mechanism more glamorous by association, we are simultaneously introducing a wonderful new allocation scheme for procedure frames. This scheme does not use a stack but instead allocates each frame with a general storage allocator. Acceptable efficiency is (hopefully) obtained by two tricks.

1) Frame sizes are quantized in some convenient way (perhaps increments of 10%) so there will not be too many different ones.

2) A list of free blocks of each frame size is kept. When a new frame of size n is needed, list n is first examined to see if it is non-empty. If so, the frame can be obtained immediately (in two instructions). If not, a more expensive, but hopefully infrequently used procedure must be invoked. Freeing a frame requires nothing more than splicing it onto the proper list. This also requires two instructions, and only the first of them needs to know which list is involved. By putting this instruction in the -1 word of the frame, we reap two benefits:

1) It is not necessary to keep track of the frame size;

2) More elaborate deallocation procedures can be spliced in by replacing the instruction with a call to some suitable routine.

An unresolved problem is how to prevent frames from existing without any references to them, or conversely, references without any frames. A reference count scheme adds some cost to most xfers, in return for which it gives no new capabilities but only an error check. Maybe this is not a real problem.

5. Inport types

The basic xfer mechanism is the same for all control transfers. The actions required inside each module to allocate or free storage, establish context and keep track of arguments, depend on what the programmer wrote, however. Thus a return, a procedure entry and a port call all have quite different internal bookkeeping. These differences are accomodated by the code which is executed within a module before a transfer and by the code at the transfer location. The transfer interface itself is the same for all types of transfer, so that any one can be connected to any other.

It is, however, necessary to be able to go from an inport to a description of the context (see below) to which it corresponds. This is done by a function which takes an inport and rummages around in the structure to which it points.

By the description of a context we mean:

1) The code segment for the context;

2) The program location within that segment;

3) A list of typed pointers to the records which contain the variables accessible in the context. Unfortunately this is not very well defined and needs further thought;

4) The owner, signal path, key, return, and current outports defined in section 3.

6. Examples

In this section we will see how to model ports, procedures and signals using the ideas developed above, and in the next section we will see how to encode these models on the 1Ø.

Notation: if an inport has transfer location tl and one parameter p, we write it as (p,tl). We assume that a frame can be given control (i.e. can be a transfer location) and that it then sets up the context it knows about and sends control to the first parameter of the port addressed by o.

A port is a pair (inport, outport). The inport is (procret, process), where process is the frame for the process which owns the port, and procret is global code which transfers to the pc saved in the process frame. The frame 'gets control' when the inport is used; it sets up the context and sends control to procret, which sends it to process.pc. The outport, of course, is a pointer to the connected inport. Then portcall (port, msg) is just

```
            process.pc ← retloc;

            xfer (port.out, address (port.in), msg);

retloc:     o.out ← i;
```

The argument pointer is in a. If desired, it can be stored in a message buffer associated with the port. The port through which control returned can be identified by its address, which is in o.

This sequence sets up the context for the process which owns the port. If the caller is some other process, it will have to do some more work to set up its own context. An example of this is given in section 7.

A simpler kind of port which carries its own pc is closer to the spirit of the basic mechanism (whether therefore better is unclear). The inport is just (pc, frame) and its semantics is:

```
            port.in.param ← retloc;

            port.in.tl ← frame;        if necessary

            xfer (port.out, address(port.in), msg);      as before

retloc:     port.out ← i:
```

and the argument pointer is sitting in a. This is O.K., since control only comes to retloc through this port.

Procedures are messier, since storage allocation is involved and the call and return are not symmetric. A procedure entry inport is (entry point, static storage segment) and a return inport is (pc, frame). Each frame has room to store an inport and also keeps track of the static storage and perhaps of other context. Then call(link, args) is

```
            frame.inport ← (retloc, frame);    this is the return inport

            xfer (link, address (frame.inport), args);

retloc:     ...
```

At the entry point we have

```
            makeframe (size);

            initializeframe (static storage segment [, other context])

            frame.retport ← i;
```

and return(results) is

```
freeframe ();

xfer (frame.retport, nil, results);
```

This is a little shady, since the results will usually be in the frame which has just been freed. The proposed fix is to reallocate the frame if anything which might demand storage is done during the storing of the results. This is quite reasonable, since the compiler knows exactly what is happening when it constructs the code to accept the results, except for the possibility of a fault during the xfer. I am not sure what to do about that. The alternative is to free the frame after storing the results, rather than in the return sequence, and that has its own set of problems: ineffeciency, and an unpleasant involvement of the caller in the internal business of the called procedure. Of course a garbage collector would take care of everything.

Signals are messier still, because of the binding algorithm embedded in their definition and because of the complications of unwinding useless frames. We deal first with signals generated by an explicit call of SIGNAL. Recall that every context has a signal path outport. The algorithm is

```
          PROCEDURE signal (code, msg)

          target ← nil

loop:     FOR p ← self.returnport, (p.signalpath if code ≠ unwind else
          p.returnport) WHILE p ≠ target DO

               self.signalpath ← p.signalpath     % bypass p if

                    catchprase generates signals %

               CASE offersignal (p, code, msg)

               =resume:  RETURN unless code = unwind else error

               =reject:  IF code = unwind THEN

                               free (p); p ← self

               % assume catchphrase requesting unwind resets inport

                    of anchor context %

                    =unwind:  target ← p:=self

                    code ← unwind
```

```
                ENDCASE

          ENDFOR

          IF p ≠ nil THEN xfer (p, nil, nil) % exit to anchor

                context of unwind %

          otherwise propagate signal somewhere else
```

Note that this code uses the procedure call machinery twice: once to obtain a context in which to run SIGNAL, and once to obtain a context in which to run the catch phrase.

When a linkage fault occurs it also generates a signal. It is convenient to make this explicit by providing an intermediary procedure:

```
          PROCEDURE ControlFaultHandler (o, i, a)

          Signal (Control Fault, (o, i) % or whatever %)

% a resume means that the transfer should be attempted again %

          free (self.frame)

          xfer (o, i, a)
```

and the handler has disappeared from view.

7. PDP-10 implementation

We adopt the following convention for the state of the registers immediately after an xfer:

o in f, the frame pointer

i and a in their own registers with those names

the target port [o] in d

the left half of [o] in tl

An inport occupies a right half-word and its first parameter is in the left half of the same word.

A dseg has two points which can be transfer locations, one for procedure entries and one for port entries, more or less:

*port entries here

```
-3(d):   movi d,frame         ;frame is set up by a port call

         movi tl,pc           ;likewise pc.  These are the process frame and
                              pc

         hrlm i,Ø (f)         ;railroad switching

*procedure entries here

Ø(d):    movi c,codebase

*load additional module-wide base registers here

         jrst Ø(tl)           ;recall  tl  has port, param if control enters
                              at Ø
```

We make use of a trick which encodes a few bits of parameter in jump addresses by duplicating the beginning of the code jumped to once for each parameter value. We also assume that we keep only one frame pointer and that if it is used for pushes or pops the compiler keeps track of how much it has moved.

A procedure call has two in line instructions (plus 1/argument):

```
...

         push f,argn

         jsp tl,call[n]             ;n is length of argument record

         framestart outport         ;opcode = f-start of the frame


call[n]: movi i,-n(f)         ;back  up  over  arguments to get location for
                              inport

         jrst call


call:    hrli d,l(tl)         ;create return inport = (pc,d)

         movem d,Ø(i)         ;and store it

         movi a,l(i)          ;set up the argument pointer

         move f,@Ø(tl)        ;pick up o

*These   three   instructions   are   the same for all the xfer sequences given
here.

         move d,Ø(f)          ;pick up target port [o]
```

```
        hlrz tl,d             ;unpack its parameter into tl

        jrst Ø(d)
```

At the entry point:

```
        jsp tl,frame[m]       ;m is the desired frame size


frame [m]:skipg f,@list[m]    ;see below for frame format

        jsp f,listempty

        exch f,list[m]

        movem i,Ø(tl)         ;save i for return

        jrst Ø(tl)            ;and go to code body
```

This assumes that frames are chained together in word Ø, with a header in list[m] for all frames of size m. We also assume that word -1 of the frame contains one of the two instructions required to restore it to list [m]. Then a return is

```
        jrst ret[n]           ;n = length of result record +1


ret[n]: movi f,-n(f)          ;f ← start of frame

        jrst ret


ret:    movi a,1(f)

        xct Ø,-1(f)           ;normally:  exch f,list [m]

        exch f,-1(a)          ;finish splicing back frame and pick up
                              outport
```

*and the three standard instructions

Timing is 33 for call, 22.5 for return or 55 total. The call can be cut to one instruction at the expense of about 15 us and pre-emption of the user UUO mechanism.

A port call is quite different, since the state has to be saved in the process owning the port. We need one word for the port: (outport, dseg-3). This word also serves as the inport for the return, which has no parameter. There are again two in-line instructions:

...

```
    push d,argn                 ;the process has only d, no f

    movi i,outport              ;set up i pointing to return inport

    jsp t1,portcall[n]

portcall[n]: movi a,-n(d)       ;back up over arguments

        jrst portcall


portcall: move t2, Ø(i)         ;pick up port

        hrrm a,Ø(t1)            ;save d

        hrrm t1,1(t1)           ;and pc in process change

        hlrz f,t2               ;extract o from port

*and the three standard instructions
```

Timing is 31us one way.

A port call to an external port (one owned by a process whose dseg is not that of the caller) is messier, since the sequence above sets up the wrong context.

```
    ...

    push d,argn

    movi i,outport

    jsp t1,xportcall[n]


xportcall[n]: movi a,-n(f)

        jrst xportcall


xportcall: movem d,-1(a)        ;save d

        movem t1,-2(a)          ;and the pc in the frame

        movi t1,xportret
```

and continue from port call +1. Finally on return control will come to xportret:

```
move f,d

move d,-1(f)

jrst @-2(f)
```

Timing is 45 one way.