Published: 03/07/67

Identification

Implementation of EPL Blocks
D. B. Wagner and M. D. McIlroy

Purpose

The PL/I definition of a block is, "...a collection of statements that defines the program region--or scope--throughout which an identifier is used as a name. It also is used for control purposes" (IBM form C28-6571-3, p. 19, which should be seen for a detailed discussion of blocks from the point of view of the language). A block may be an external procedure, an internal procedure, or a begin block. The most important fact about the implementation of blocks is that each generation of each block has a corresponding frame in the stack. This section describes the mechanisms used for initializing and terminating these blocks, and for accessing variables of containing blocks.

External Procedures

Ignoring the fact that segments may be bound together (for which see The Binder, MSPM BX.14.01) external procedures correspond one-for-one with procedure segments. Every PL/I program segment is an external procedure with one or more external entries. The first label on the first procedure statement in the program segment is considered the "procedure name" and the "primary entry point"; in EPL (but not in PL/I) this name is used as the name of the procedure segment.

An external entry is of course called using the standard call-save-return sequences described in BD.7.02-03 (with one trivial change noted below). An entry beta in segment alpha is referenced using the notation

```
alpha$beta
```

and the simple name beta with no "$" refers to

```
beta$beta
```

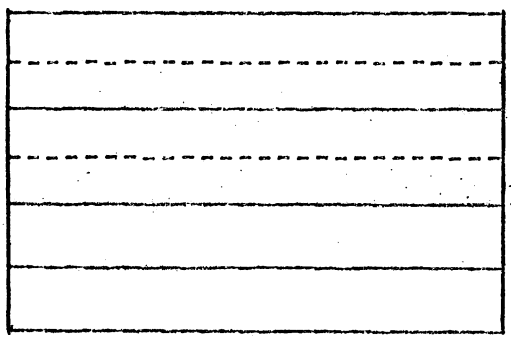
Internal Procedures

In EPL internal procedures are called using the standard call-save-return sequences with the argument list modified by attaching to it the value of the stack pointer for the embracing block. This modification is described in BD.7.02. This implementation is mandatory for internal procedures that may be called from another segment. As an example consider the statements:

```
a: begin;
b: proc;
   . . .
```

A call to b (no matter whence) includes in the argument list the stack pointer for the generation of storage for a. (of course a must be active at the time of the call to b. This stack pointer is used by b for accessing variables in embracing blocks.

This situation is complicated by the fact that an internal entry may be passed as an argument, and thus may be called from procedures which have no way of knowing the stack level of its embracing block or even whether it is internal or external. To solve this problem, whenever an entry is passed as an argument, the object passed is a six-word block (identical to a label) as follows:



Whenever any entry parameter is called, the caller includes the stack pointer value from the entry parameter in the same way as it would be passed in a call to an internal procedure. External procedures ignore this extra, while

internal procedures use it to create a display without caring about its source.

Begin Blocks

In EPL a begin block is implemented precisely as an internal procedure, so that the statements

```
a:  begin;
    . . .
    end a;
```

might compile to something like

```
a:    . . . (set up argument list for no arguments,
            but with value of sb<--sp attached)
      call xx0239 (argument list)
      tra  xx0240
xx0239: save
      . . .
      return
xx0240: null
```

The display

As was mentioned earlier, an internal procedure or begin block receives attached to its argument list the value of the stack pointer for its statically embracing block. In EPL it uses this to enable it to refer to outer blocks by setting up the display, which is a list of ITS pairs giving stack levels for all statically embracing blocks. The display starts at a fixed location in the stack frame for any internal block and for a block at level i (where an external procedure is at level 0) is 2*i words long. To refer to a variable at location a in the stack frame for an embracing block n levels back, the code might then be:

```
eapbp     sp|display+2*n-2,*

lda       bp|a
```

When an internal procedure or begin block at level i is entered, it sets up its display by inserting the stack pointer attached to the argument list into the display and then appending a copy of the display from the embracing block (which is 2*i - 2 words long).

Prologues, Epilogues, and the Non-Local Go To

Each block (external or internal) begins with a prologue and ends with an epilogue. The prologue performs various initialization tasks such as setting up the display, creating specifiers and (sometimes) dope for automatic variables, etc. It is not terribly important in this discussion.

The epilogue performs a number of tasks which must be done when a block is terminated. These include:

1) Reverting on-conditions
2) Freeing the storage occupied by automatic varying strings
3) Popping up the epilogue stack

The problem with epilogues is shown by the following series of statements:

```
a:      begin;
q:      . . .
b:      begin;
        . . .
c:      begin;
        . . .
        go to q;
        end c;
        end b;
        end a;
```

The statement "go to q" is a non-local go to: when it is executed, the stack level must be brought down to the level of a; furthermore, the epilogues for both of the blocks c and b must be performed.

For the benefit of the non-local go to, a push-down list of epilogues to be performed is kept in a static segment. In location

```
< trap_ >|[ epi]
```

is a pointer to what is called the current epilogue handler. The value of this pointer is initially null. An epilogue handler has the form of the following structure:

```
dcl  1 epilogue_handler ctl (p),
     2 loc ptr,           /* location of epilogue */
     2 stack ptr,         /* stack level of block */
     2 back ptr;          /* to previous epilogue handlers */
```

This piece of data is normally located in the stack frame for the block to which the epilogue belongs.

The prologue for a block with an epilogue executes the following sort of code:

```
dcl my_handler auto like epilogue_handler;
my_handler.loc = . . .;  /* location of epilogue */
my_handler.stack = . . . /* current sb←sp */
my_handler.back=trap_$epi;
trap_$epi_ = addr (my_handler);
```

At the end of the epilogue, the following statement serves to revert to the previous epilogue handler:

```
trap_$epi_ = my_handler.back;
```

The epilogue is a sequence of code which performs its task, reverts the epilogue pointer trap_$epi_, and finally executes a return sequence.

Because of difficulties with asynchronous interrupts, an attempt is made to code the epilogues generated by EPL with the following doctrine:

> "It will be harmless to execute all or part of an epilogue more than once".

The chain of epilogue handlers described above is used by the run-time routine synep_ which performs a so-called "synthetic epilogue" for the non-local go to. Synep_ is described in BN.7.03.

EPL's Variation of the Save Sequence

There are two problems with the save sequence presented in BD.7.02: it does not permit stack frames longer than 16k words, and it does not permit use of a single copy of the code (which is six words long) by every procedure and block in a program. Thus the following slight variation of the save sequence is compiled by PL/I:

```
      eax7  t     load index register 7 with length of next frame
      tsx0  .sv   to save subroutine
      . . .
.sv:  eapbp  sp|18,*
      stpsp  bp|16
      stpap  bp|26
      eapap  bp|0,7
      stpap  sp|18
      eapsp  bp|0
      stb    sp|0
      tra    0,0
```

Since EPL never uses the entrance value of ab◄—ap except as it is stored in sp|26, it is unimportant that this save sequence "clobbers" that base pair.

The instruction above,

```
stb  sp|0
```

saves the value of the base pair lb◄—lp for use by internal procedures in case they are entered from outside the current segment.