Published: 8/31/66

Identification

Sequential Linear I/O
J. F. Ossanna, V. A. Vyssotsky, G. G. Ziegler

Purpose

The Multics I/O system provides capability for sequential linear I/O. This section describes in detail the I/O system calls for performing sequential linear I/O.

Sequential Linear Frames

An existing linear frame may be attached to a process (or a new linear frame may be created and attached to a process) as a sequential frame by an attach call to the I/O system (see section BF.1.02). A frame which already exists when it is so attached may previously have been attached to processes as either a sequential or random frame, or both. The primitive operations available for transmitting data to and from a linear frame attached as a sequential frame are discussed in what follows.

The Write Call

A sequence of elements may be written into a sequential linear frame by means of the write call, whose general form is:

call write (name,elemno,workspace,nelem[,status])

The argument name is a character string of 1 to 31 characters. Its content is either a streamname or a frame id. If name is a streamname, it refers to the frame to which the stream is attached. The argument elemno is a 35 bit signed integer whose value must be non-negative. The value of elemno is the difference between the element number of the first element to be written and the element number of the current element. The argument workspace is a pointer to the data to be written. Specifically, in PL/I terminology, workspace is a pointer variable; the I/O system will act as if the based variable associated with the pointer variable workspace were a bit string of length nelem times the element size. The argument nelem is a 35 bit signed integer specifying the number of elements to be written. The value of nelem must be non-negative. The optional argument status is a bit string returned to the caller by the I/O system; it contains status information about the transaction. See section BF.1.21 for a description of the content of the status argument.

Suppose alpha is a newly created and attached sequential linear frame and its declared element size is 9 bits. Then the call

```
call write('alpha',,data,100,state)
```

would cause the first 100 characters of data to be written as elements 1,...,100 of alpha; on return the bit string state would contain status information about the transaction. After this call, alpha is 100 elements long and its current element number is 101. The current element is always the element just following the last element written (or read). Thus, a second write

```
call write('alpha',,other,100,state)
```

will write the first characters of other as elements 101,...,200 of alpha. If a write call is followed by another write call with elemno > 0, the second call will cause skipping over elements and/or writing of zero elements, followed by writing of the specified data. For example, consider the sequence

```
call write('alpha',0,data,100)
call write('alpha',10,other,100)
```

and assume as before that alpha is newly created. The first call will write the first 100 elements of data into alpha. The second call specifies that the first 100 elements of other are to be written into alpha in such a way that the first element of other is written into alpha ten elements beyond where the 100th element of data was written. The I/O system achieves this result by inserting ten elements of binary zeros. Thus the two calls together wrote into alpha the first 100 elements of data, followed by ten elements of binary zeros, followed by the first 100 elements of other.

The rules governing insertion of zero elements on write calls for linear frames are as follows. Suppose the actual length of a linear frame called henry is E elements. Consider the call

```
call write('henry',elemno,data,nelem,state)
```

The argument elemno specifies that writing is to begin elemno elements beyond the current element at the time of the call.

Let e be the number of the first element to be written. If e > E there is a possibly zero gap between the last element already in the file and the first element to be written; this gap will be filled with e-E-1 elements of binary zeros. Then the nelem specified data elements will be written. This will happen even if nelem = 0. For example, suppose henry is 10 elements long and the current element number is 8. Then

```
call write('henry',10,data,0)
```

will cause henry to be extended by 7 elements of binary zeros (elements 11 to 17); on return the file is 17 elements long, and the current element number is 18.

In the normal truncation mode for writing of sequential frames, writing a sequence of elements causes the last element written to become the last element of the frame. For example, suppose that when alpha is attached it is an existing frame whose content is 1000 elements of the declared size. If the first call after attachment is

```
call write('alpha',100,data,100)
```

then the first 100 elements of the frame will be skipped over, the next 100 will be replaced by 100 elements from data, and the remaining 800 elements previously in the frame are no longer there. After completion of the write, the length of the frame is 200 elements, and the current element is element number 201. (See section BF.1.04 for a discussion of write in the replacement mode.)

A request to write beyond the declared maximum length of a frame will be rejected, no data will be written, and an end-or-frame status will be returned to the user. Note that this differs from a read which requests data extending beyond end-of-frame. A read request for 10 elements of data when only 6 remain in the frame will result in the transmission of the 6 elements and an end-of-data status return. A write request, however, to write 10 elements when there is room for only 6 will be rejected with no data at all written.

The Read Call

Elements may be read from a linear frame by using the read call, whose general form is

```
call read(name,elemno,workspace,nelem[,nelmt[,status]])
```

The arguments of read are the same as the corresponding arguments of write, except that the read call has an additional optional argument, nelmt. This argument is a 35 bit signed integer, returned by the I/O system. Its value is the number of elements transmitted from the frame to the caller's workspace, and lies in the range $0 \leq$ nelmt $\leq$ nelem. If no data is read, nelmt = 0. If the read request is completely fulfilled, nelmt = nelem. If the remainder of the frame is less than nelem elements long, or if a break element stops transmission, the value of nelmt shows how may elements were actually transmitted. As an example, suppose that the frame attached to alpha is positioned at the beginning of frame, either immediately after attachment or immediately after a first call. Suppose further that the element size is 9 bits and that the frame is 1000 elements (characters) long. Then the call

```
call read('alpha',0,data,100)
```

will cause the first 100 characters of the frame to be read into data. After return from the call the current element is element number 101.

If it had been desired to read the first 50 elements of the frame into data, skip the next 50, and read elements 101 to 150 into other, it could be done by the sequence

```
call read('alpha',,data,50)
call read('alpha',50,other,50)
```

If a read call requests data from a place partly or wholly beyond the end of the data in the frame, any of the requested data which exists will be transmitted, and the status return will show that less data was transmitted than had been requested. For example, assume again that frame alpha contains 1000 elements, and suppose the current element 999. Consider the sequence

```
call read('alpha',,data,1,count,state)
call read('alpha',,other,1,count,state)
call read('alpha',,any,1,count,state),
```

The first read will transmit element number 1000 to data, and will return a last-data status. The second read will not transmit data, and will return end-of-frame status. The third read will not transmit data and will return beyond-end-of-frame status.

The Current Element

After a successful write call, the next element to be written is known as the current element.

Specifically, a sequential linear frame has a current element in the following cases. If the most recent read, write, seek or delete call referenced an element of the frame (i.e. was not rejected by the I/O system) and if no subsequent first or tail call has occurred, then the element beyond the last element referenced by that most recent read, write, seek or delete call is the current element. After a frame is initially attached, or immediately after a first call, the current element is the first element of the frame. After a tail call the current element is LAST+1. For a precise definition of current element, see section BF.1.10.

The Tell Call

It is often useful to be able to determine the current element number of a linear frame. This can be accomplished by means of the tell call, whose general form is

call tell(name,elemno[,status])

The arguments name and status are the same as the corresponding arguments of a write call. The argument elemno is a 35 bit signed integer. The value of elemno at time of the call will be ignored and overwritten by the I/O system. At time of return, elemno will contain the current element number for the indicated file, unless the call was rejected (bit 4 or bit 15 of status set to 1). A call to tell does not change the current element number, nor does it change the data content of the frame. On return from a call to tell the value of the argument status will be exactly what it would have been if the call had been a call to seek with elemno = 0.

The Seek Call

The seek call allows an element to be designated as the current element without causing transmission of data. Its general form is

call seek(name,elemno[,status])

and its arguments are the same as the corresponding arguments of write. If stream alpha is attached to a frame which has a current element, then

call seek('alpha',n)

will cause the current element to be the nth element after the one which was current before the call.

If the new current element number resulting from a seek call is beyond the last element of the frame, the current element number still exists, although no current element exists. In this case the status return from the seek will show end-of-frame. If, however, upon initiation of a seek call the current element number is already one or more greater than the number of the last element of the frame, the status return from the seek will show beyond-end-of-frame. A write request after such a seek will be handled normally, provided that the seek call did not increase the current element number beyond the declared maximum size of the frame.

The Breaks Call and Reading with Breaks

Any particular set of elements may be defined as break elements; a break element will stop reading of data on a read call. When a linear frame is first attached, there are no break elements. Break elements may be declared by a call whose general form is

```
call breaks(name,breakptr,nbrks[,status])
```

The arguments name and status are the same as the corresponding arguments of the write call. The argument breakptr is a pointer to a set of break elements. Specifically, in PL/I terminology, breakptr is a pointer variable; the I/O system will act as if the based variable associated with the pointer variable brkptr were a bit string of length nbrks times the element size. The argument nbrks is a 35 bit signed integer specifying the number of break elements in the set; the value of nbrks must be non-negative. Each of the elements in the set specified by a breaks call will be treated as a break element until another breaks call occurs. Note that length of the break must be the same as the element length.

Duplicate elements are permissible in the break set. For example, suppose the element size on stream beta is five bits, and consider the sequence

```
declare sam bit(20)initial('10101010101111101010'B);
declare p pointer;
p=addr(sam);
call breaks('beta',p,4);
```

After the call to breaks, the elements 10101, 01010, and 11111 will be break elements for stream beta until another breaks call is given for stream beta. A read call on a stream connected to a linear frame will not transmit any elements beyond a break element transmitted by the

read call. More specifically, consider any read call for a linear frame

```
call read(name,elemno,workspace,nelem,nelmt,status)
```

If none of the nelem elements to be read is a break element, then nelem elements will be transmitted to the workspace. If, however, one or more of the nelem elements to be read is a break element, the elements up to and including the first such break element will be transmitted, and the remaining elements (if any) will not be transmitted. Suppose, for example, that character string text is being read on serial linear stream alpha, that the break characters are , and ; and that the text, starting with the current character, is

```
bbcd,e;,;fghi,...
```

Consider the sequence

```
call read('alpha',,data1,3)
call read('alpha',,data2,3)
call read('alpha',,data3,3)
call read('alpha',,data4,3)
call read('alpha',,data5,3)
call read('alpha',,data6,3)
```

Then the character bbc will be read into data area data1, the characters d, into data2, the characters e; into data3, the character , into data4, the character ; into data5, and the characters fgh into data6.

The Delete Call

The general form of the delete call is:

```
call delete(name,elemno[,status])
```

and its arguments have the same form as the corresponding arguments of the write call. The effect of a delete call for a linear frame in the truncation mode is exactly the same as the effect of a write call with nelem = 0. If alpha is an existing frame of 1000 elements and the current element number is 300, the request

```
call delete('alpha',100,state)
```

will delete elements 400 to 1000. In the same way

```
call write('alpha',100,data,0,state)
```

will also delete elements 400 to 1000. (See section BF.1. for a discussion of write and delete in the replacement mode.)

The First Call

After reading or writing part or all of a sequential frame, it is frequently necessary to go back to the beginning of the frame and start reading or writing again from the beginning. This is accomplished by the first call whose general form is

call first(name[,status])

For example, suppose that a number of elements have been written in a frame attached to stream alpha, and it is now desired to read those elements.

call first('alpha')

will position the frame so that a read call can read data from the frame starting with the first element.

The Tail Call

When adding elements to an existing frame it is useful to be able to skip to the end of the current contents. This may done with the tail call whose general form is

call tail(name[,status])

Following such a call the current element number is LAST+1 and a write with elemno = 0 or null would write data immediately following the last element already in the frame. A read call of any kind would get a status return showing end-of-frame.

An Example

Let us consider as an example the problem of writing a PL/I procedure to make a copy of an arbitrary linear I/O frame in the file system. The procedure will be called by:

call copy(oldfile,newfile,state)

where oldfile and newfile are character strings; oldfile is the name of a readable frame in the file system, and newfile is the name to be given to the copy to be created. State is a bit string of length 72. In our example we shall simplify error handling by passing the buck up to the caller if any difficulty arises in copying the frame. Code to perform the copy operation is shown in figure 1.

The code works as follows. First we attach the frame to be copied as a sequential read-only linear frame and give it the id tempi, which will also be used as a streamname for reading the frame. If the attachment does not succeed, we return to the caller. If it does succeed, we attach the frame to be copied into as a sequential write-only linear frame and give it the id tempo, which will also be used as a streamname for writing the frame. If the attachment does not succeed we detatch tempi and return to the caller. If it does succeed, we go on to establish bounds for both frames. We want to make an exact copy, and we have no idea what element size was used to write tempi, so we use element size 1 bit. We bound the frame sizes by the actual size of tempi, thereby ensuring that tempo will hold the copy while asking for as little space as possible. Even so, we may not have succeeded in getting what we needed, so we test to find out. If not, we scrap tempo, detach tempi, and return. If the bounds were established successfully, however, we test to see whether tempi has ever been written. To do this we call first, which does nothing, since tempi is already positioned at beginning of frame, but which returns status that we can examine. If no data has ever been written into tempi, then we have already made tempo an exact copy of tempi by creating tempo and not writing anything into it, so we detach both frames and return. If, however, data has been written in tempi, we copy it into tempo 2304 bits at a time until we have copied the last data from tempi. We then detach both frames and return. We chose to copy 2304 bits at a time because 2304 bits is 64 words, which is a magic number to the file system. We could have copied any number of bits at a time; we suspect that the copying is especially efficient if we do it in 64 word chunks.

This example recurs in more elaborate forms in sections BF.1.17 and BF.1.18, where copy procedures are shown for copying more general (and less tractable) frame structures.