Published: 02/06/68

Identification

Tape Reader
N. I. Morris, M. A. Padlipsky

Purpose

When Collection 1 has been read from the Multics System Tape, the Multics Initializer is able to employ a considerably less primitive tape reading mechanism than has been used heretofore in the initialization process. This mechanism is known simply as the Tape Reader, and will be employed throughout the remainder of initialization by the Segment Loader (see BL.6.01); it is also employed by the system initializer (fs_init_4; see BL.10.03). The Tape Reader is the only Multics Initializer procedure which need take cognizance of the Multics standard format for magnetic tapes (BB.3.01); it does not take cognizance of the logical structure of the MST. The Tape Reader's role is to furnish a specified number of words from the MST (via the I/O hardware and a physical record buffer area) to an area specified by its caller; the words are taken from the MST sequentially.

Introduction

The buffer area used by the Tape Reader is segment <physical_record_buffer>. This segment is created by the Bootstrap Initializer, which uses a magnetic tape input routine similar to the Tape Reader. The organization of <physical_record_buffer> is as follows: In <physical_record_buffer>|0 (referred to as record_index, below, although the segment has no linkage definitions associated with it), a count is maintained of the number of words which have already been delivered to the user from the current buffer. Locations <physical_record_buffer>|1 through <physical_record_buffer>|272 contain the "current record buffer", and relative locations 273 through 544 contain the "next record buffer". Both the "current record buffer" and the "next record buffer" have the same structure: The first 8 words contain the physical record header, the next 256 words contain the physical record data and the final 8 words contain the physical record trailer (see BB.3.01). The role of the "next buffer" will be discussed below.

The Tape Reader comprises two distinct procedures: tape_reader and tape_io. Procedure tape_reader is responsible to its caller for the proper manipulation of the physical buffers and transmission of the data contained therein to the caller. Procedure tape_io is responsible to tape_reader (its caller) for causing the actual magnetic tape input-output operations to be performed. One way of looking at the division of labor involved is that tape_io fills <physical_record_buffer> and tape_reader empties it. It is important to note that included under the notion of proper manipulation of the buffers is the elimination (i.e. non-transmission) of "repeated records"--that is, records which were re-written owing to tape errors in writing, of which only the final, accurate version should be passed on.

Usage

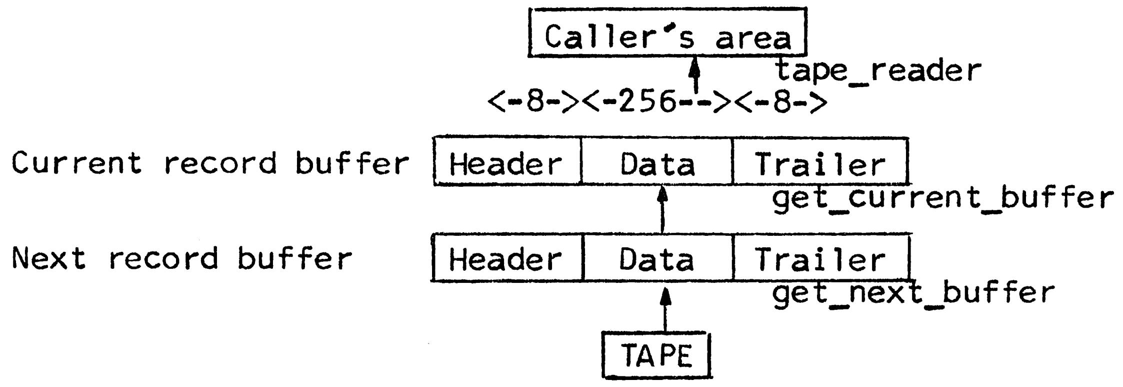
The calling sequence for the Tape Reader is:

```
call tape_reader (loc, nwords);
```

where loc is an EPL-type pointer to the base of the area into which the data are to be copied, and nwords is the number of words to copy.

Implementation

Pictorially, the Tape Reader's processing may be viewed as



where the arrows represent data flow and the names beside the arrows indicate which internal subroutine in tape_reader is managing the flow of the data. Tape_reader extracts data from the current record buffer until it is empty.

At that point, tape_reader invokes an internal procedure named get_current_buffer, which refills the current record buffer from the next record buffer and in turn invokes an internal procedure named get_next_buffer, which refills the next record buffer via the tape_io procedure. If the call to get_next_buffer proves to have brought in a repeated record, get_current_buffer writes the later version (in the next record buffer) into the current record buffer and calls get_next_buffer again (details are given below).

The logic is as follows:

1. Calculate a, the number of words left in current record buffer. The physical record header portion of the current record buffer contains a bit count of the length of the data record; call it nbits. Then, nbits divided by 36 minus record-index (the running count of words already copied from the current buffer, maintained at <physical_record_buffer>|0) gives the number of as-yet-uncopied words left in the current record buffer.

2. Any words left? If a is greater than zero, there are meaningful data left in the current buffer; proceed to step 4.

3. Refill current buffer. At this point, there are no words to copy left in the current record buffer; therefore, call the internal routine get_current_buffer. On return, set record-index to zero and a (the number of data words now in the buffer) to nbits divided by 36 (as in step 1).

4. Adjust counts. If n, the number of words remaining to be copied (i.e., the current, possibly already decremented, value of nwords), is less than a, set a to that value; that is, a is set to the lesser value of a and n. Next, decrement n by a and set the result into n; that is, subtract the number of words about to be copied from the running total of words needed to be copied.

5. Transmit. Call the internal routine move, to cause a words to be copied through the loc pointer at the appropriate index values.

6. Done? On return from <u>move</u>, if there are more words to be transmitted ($\underline{n}>0$), transfer to reader_loop (step 3). Otherwise, update record_index (<physical_record_buffer>|0), and return.

<u>The internal routine get current buffer</u>:

The calling sequence is

```
call get_current_buffer;
```

The logic is as follows:

1. Move the contents of the next record buffer into the current buffer.

2. Call the internal routine get_next_buffer. This routine will invoke the external procedure tape_io appropriately, and on return from it the next data record from the MST will be in the next buffer.

3. At this point, it is necessary to investigate the record number of the new record, in order to determine whether it represents a repeated record. If the record number of the data record in the next record buffer is equal to the record number of the data record in the current record buffer, transfer back to step 1 and get another new record (because the record in the next record buffer is a re-writing of the one in the current record buffer).

4. If the record number of the data record in the next record buffer is one greater than the record number of the data record in the current record buffer, return. (The current record is valid.)

5. If the record numbers are neither equal nor sequential, an error condition exists. This error is fatal; the current initialization run is terminated by a call to <u>panic</u>.

<u>The internal routine get next buffer</u>:

The routine is called to manage the reading of a physical record from the MST into the next record buffer of <physical_record_buffer>. (Upon return from get_next_buffer, the next record buffer will contain either another copy of the record in the current record buffer, in which case get_current_buffer must call get_next_buffer again, or the next record in sequence after the current record.) The logic is as follows:

1. Initialize try. Set the variable try, which is a count of the overall number of attempts to read the tape, to zero.

2. Initialize n. Set the variable n, which is a count of the number of records read this try, to zero.

3.a. Increment n by 1.

  b. Call tape_io$read to read one record from the MST into the next record buffer.

4. Check status. If the tape was not ready, return to step 3b. If an end of file mark was encountered, go to step 12. If a device data alert was encountered, return to step 3a. Otherwise, proceed to step 5.

5. Validity checks. The first and eighth words of the header and the trailer and the checksum must be checked for validity (see BB.3.01 for tape format). If not valid, return to step 3a.

6. If an administrative record was encountered which is not an EOR record, return to step 3a.

7. If the record number of the record at hand is the same as the record number of the record in current_record_buffer, proceed to step 10. If the former number is less than the latter, this implies that the tape has been backspaced too far; return to step 2.

8. If the next record number is greater than the current record number plus one, this implies that the tape has been spaced along too far; proceed to step 14. Also, if n minus files (a count of the number of EOFs encountered) minus one is not equal to the repetition number of the current record, proceed to step 14 for backspacing.

9. Return. The checks in steps 4 through 8 having been passed, the routine returns to its caller with a new record in the next record buffer.

10. At this point, the record numbers of the next and current records are the same. It is necessary to confirm the fact that the repetition numbers are correct before returning; therefore, if the repetition number of next record is less than or equal to that of current record (implying that the tape has been backspaced too far), return to step 2. Otherwise:

11. Return (next_record_buffer contains a repeated version of current_record_buffer).

12. At this point in the logic, an end of file mark has been encountered. If n equals one, set file to one. (More than one EOF should never be encountered; therefore, unexpected ones are ignored.)

13. Return to step 3a.

14. At this point, the tape must be backspaced unless the allotted number of tries has been exceeded. Therefore, increment try by one; if try is now greater than ten, call panic (see step 5 of get_current_buffer.) Otherwise, call tape_io$backspace n+2 times. (The "extra" two times are included at attempt to insure that the tape is backspaced at least enough to try again; empirically, it turns out that certain sections of bad tape will require this sort of treatment, as the apparent number of records encountered when going forward can be greater than the apparent number of records encountered when going backward.)

15. Return to step 2.

tape_io

It is intended that the reader of this section be familiar with the operation and use of the GIOC (see G0050) and the High-Performance Common Peripheral Channel (HPC).

Tape_io is a procedure called by tape_reader to do all magnetic tape I/O operations. The phase 1 version of tape_io issues all connect operations itself. It does not use the GIM. The following entries are provided in tape_io:

1. read (ptr, count, status): read binary tape

2. backspace (status): backspace tape one record

3. rewind (status): rewind tape

4. unload (status): rewind and unload tape

5. skip_file (status): forward space tape one file

where ptr is a pointer to tape buffer

count is the number of words to be read from tape

status is a fixed binary number to contain the major status of the tape controller after the i/o operation is performed.

Tape_io assumes that upon first entry, the GIOC mailbox segment <mailbox> will contain a COW at location 24(10) and a CCW at location 14(10); the CCW contains the channel number and the device number in use by the Multics Initializer. The tape_io routine will use connect channel 8 for all connect operations and status will be returned through status channel 1. All operations are performed in multiple-physical instruction (MPI) mode. Emergency status will cause the GIOC bell to ring. Since the DCW list for all I/O operations and the status queue are contained within tape_io, it is an impure procedure. It must be wired down.

The sequence of operations for all I/O requests is shown below:

1. Set up CPW in <mailbox>|8.

2. Set up CCW in <mailbox>|14. Insure that the MPI mode is set. Save the channel number (c) from CCW.

3. Set up the LPW in <mailbox>|2*c.

4. Set up the DCW list inside tape_io.

5. Pick up and discard status words until status queue is empty.

6. Issue connect to <mailbox>|0.

7. Wait until terminate status or external signal status appears in status queue.

8. Return major status and return to caller.

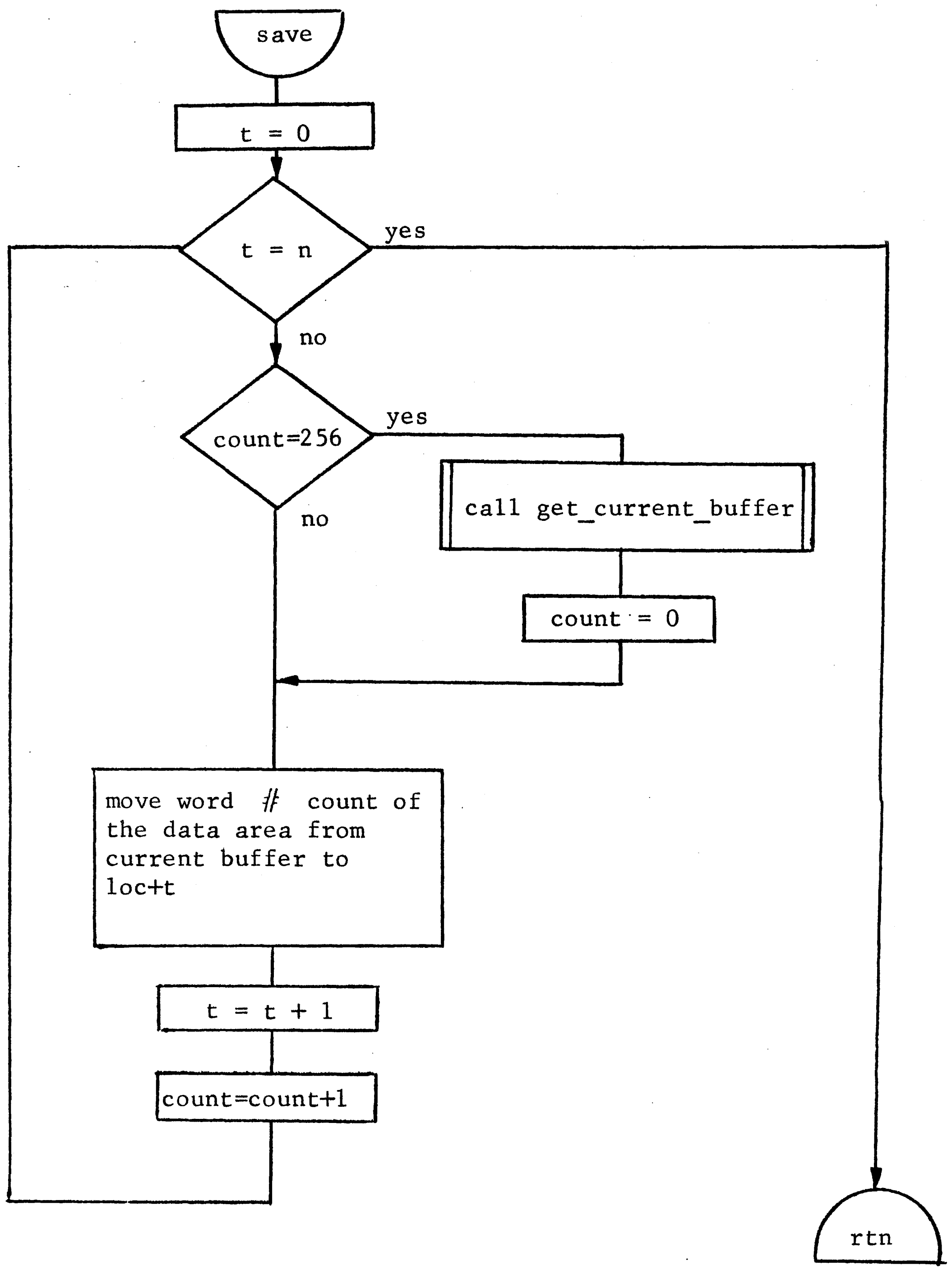


Figure 1: Tape reader

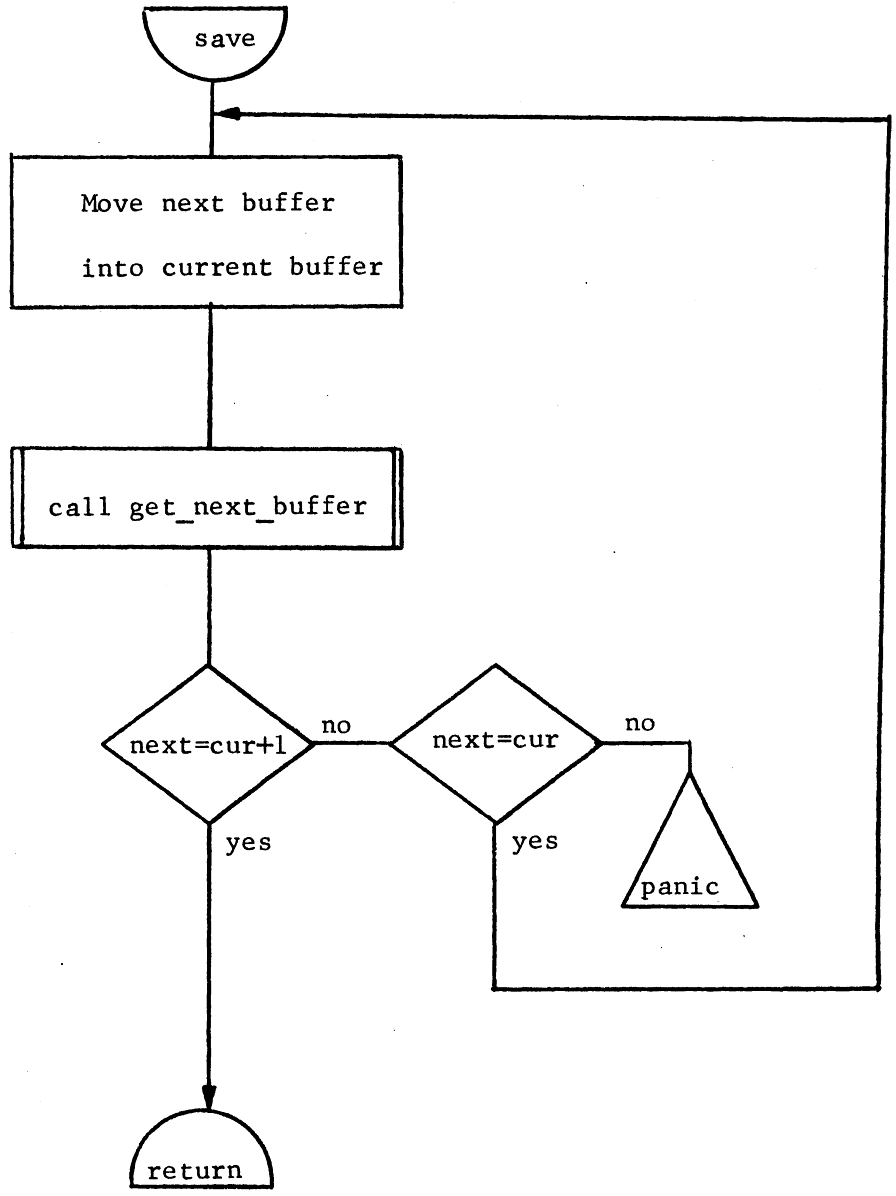


Figure 2: Get Current Buffer

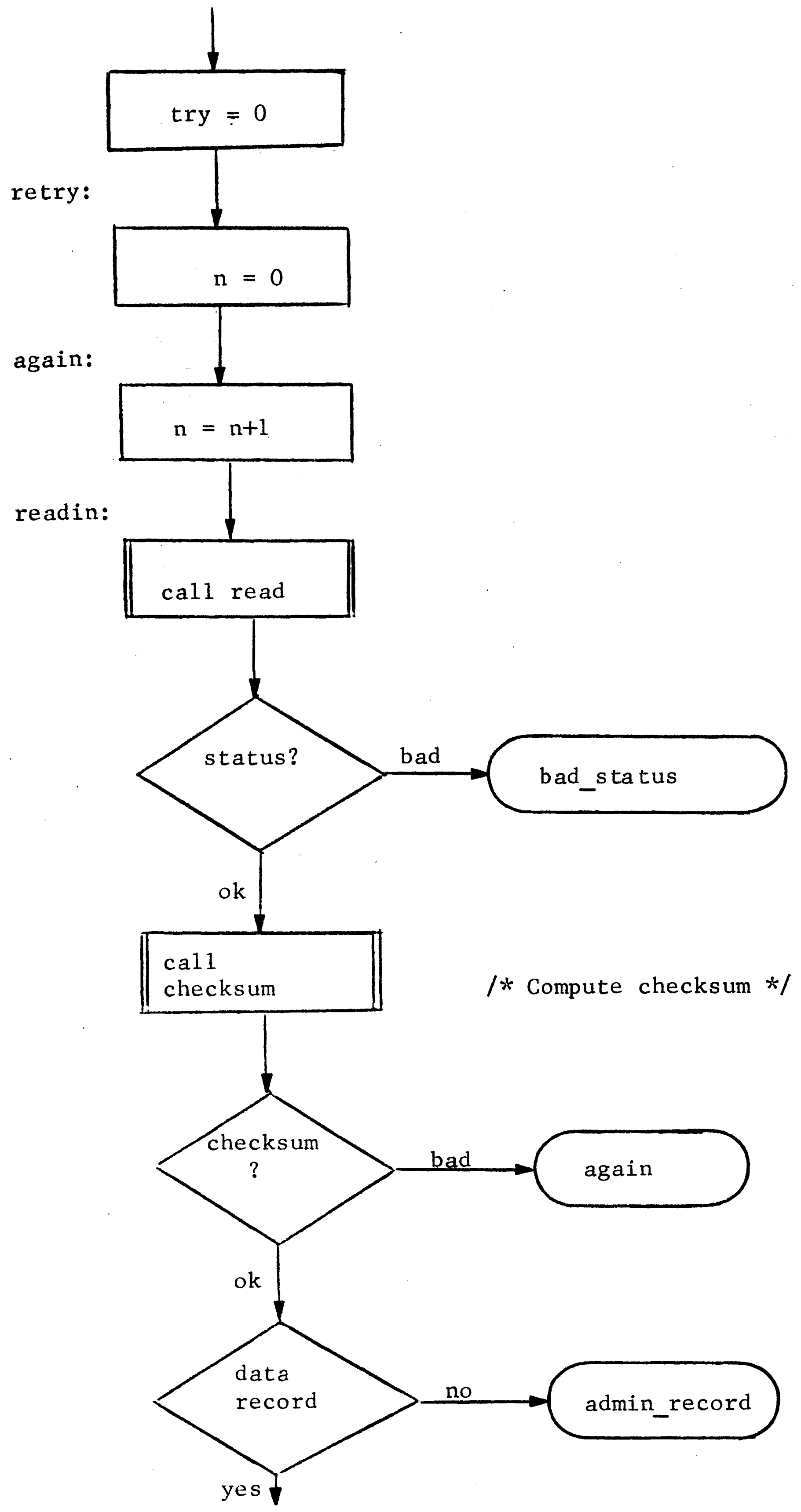
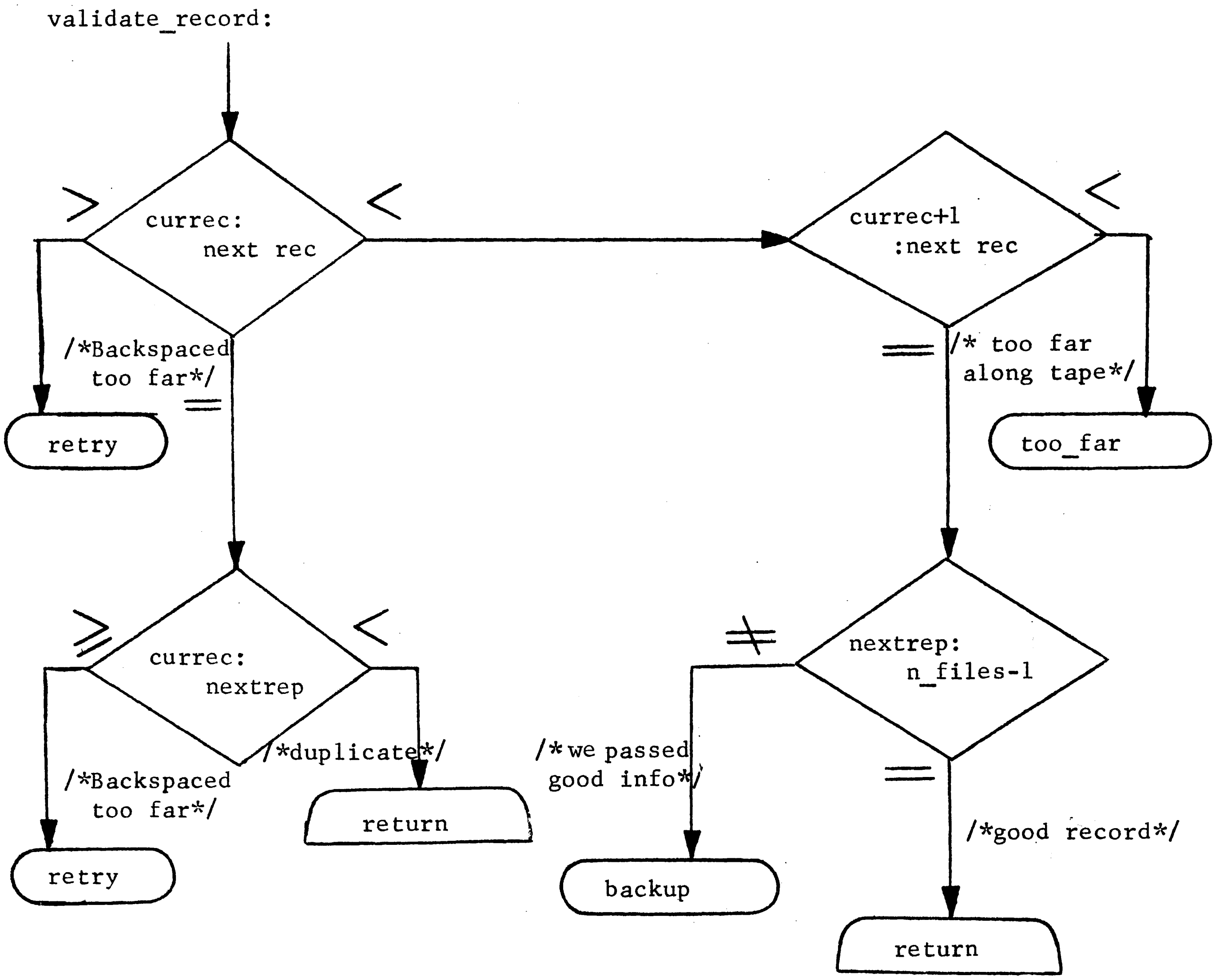
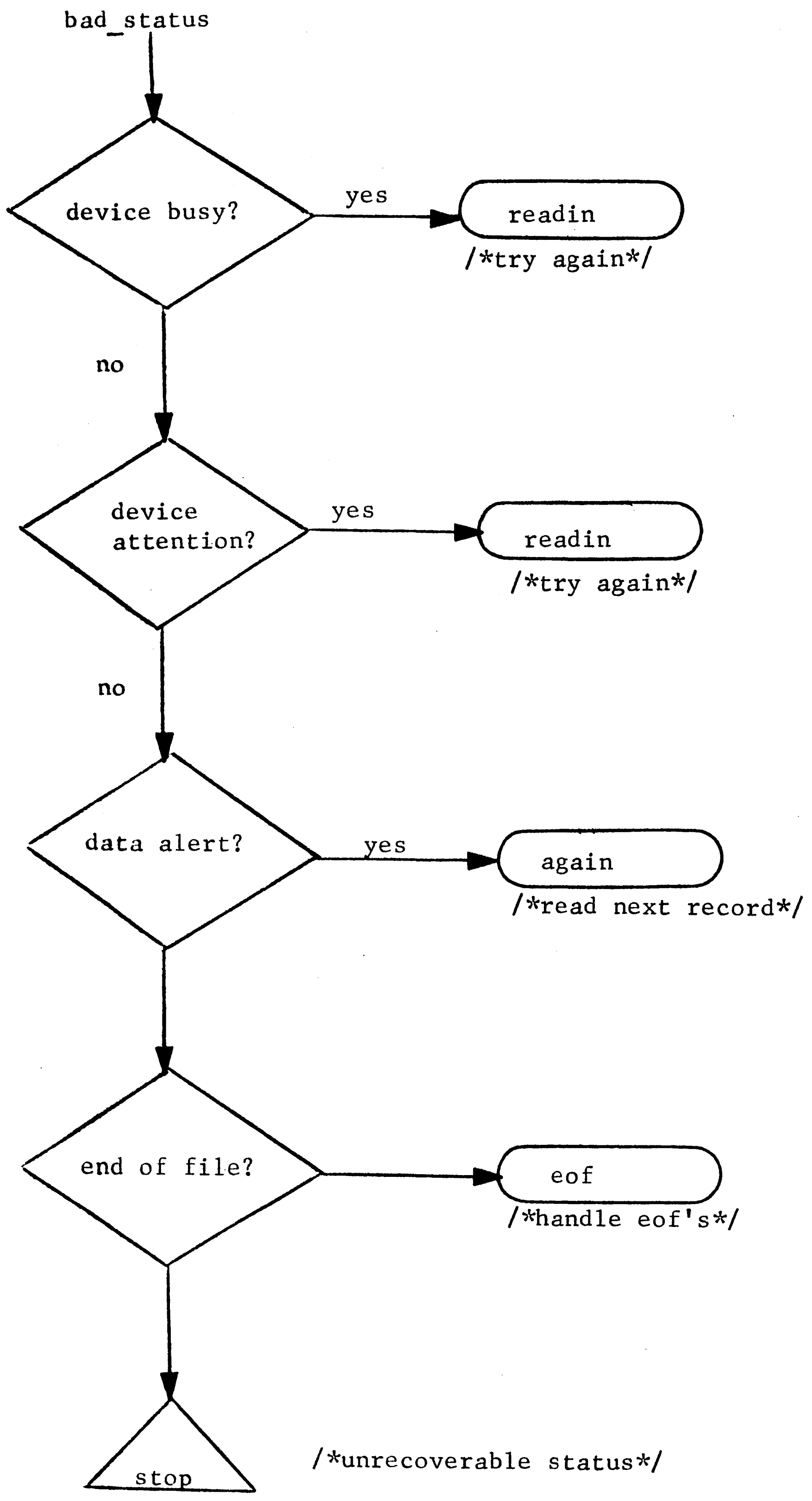


Figure 3: get_next_buffer

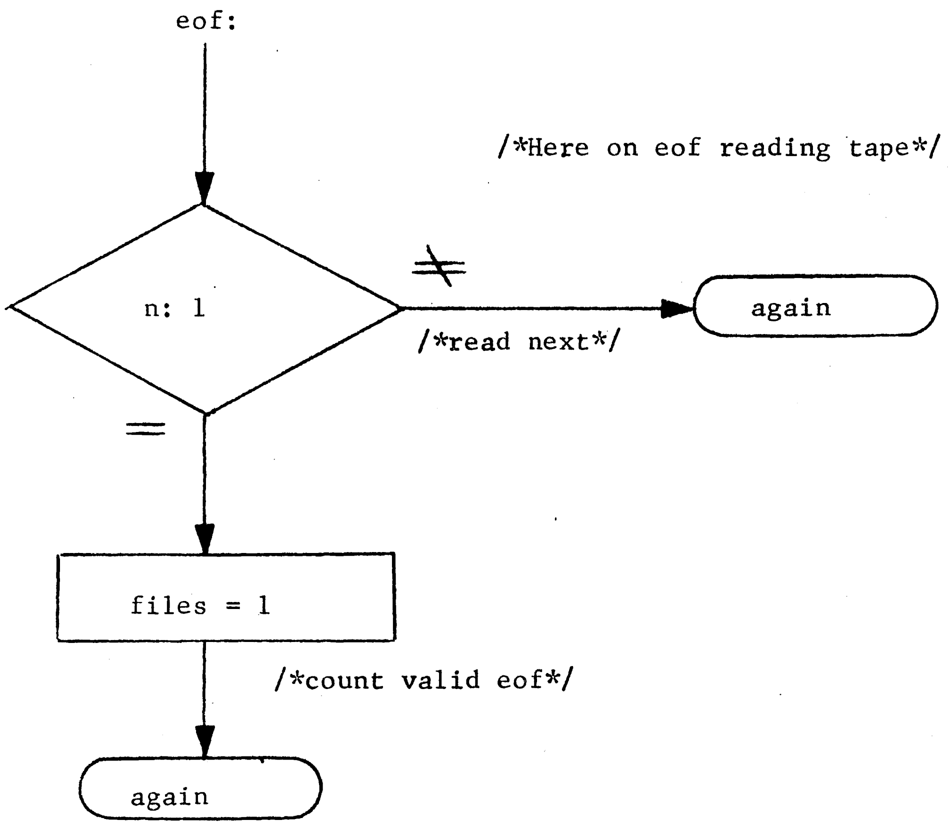
continuation of figure 3



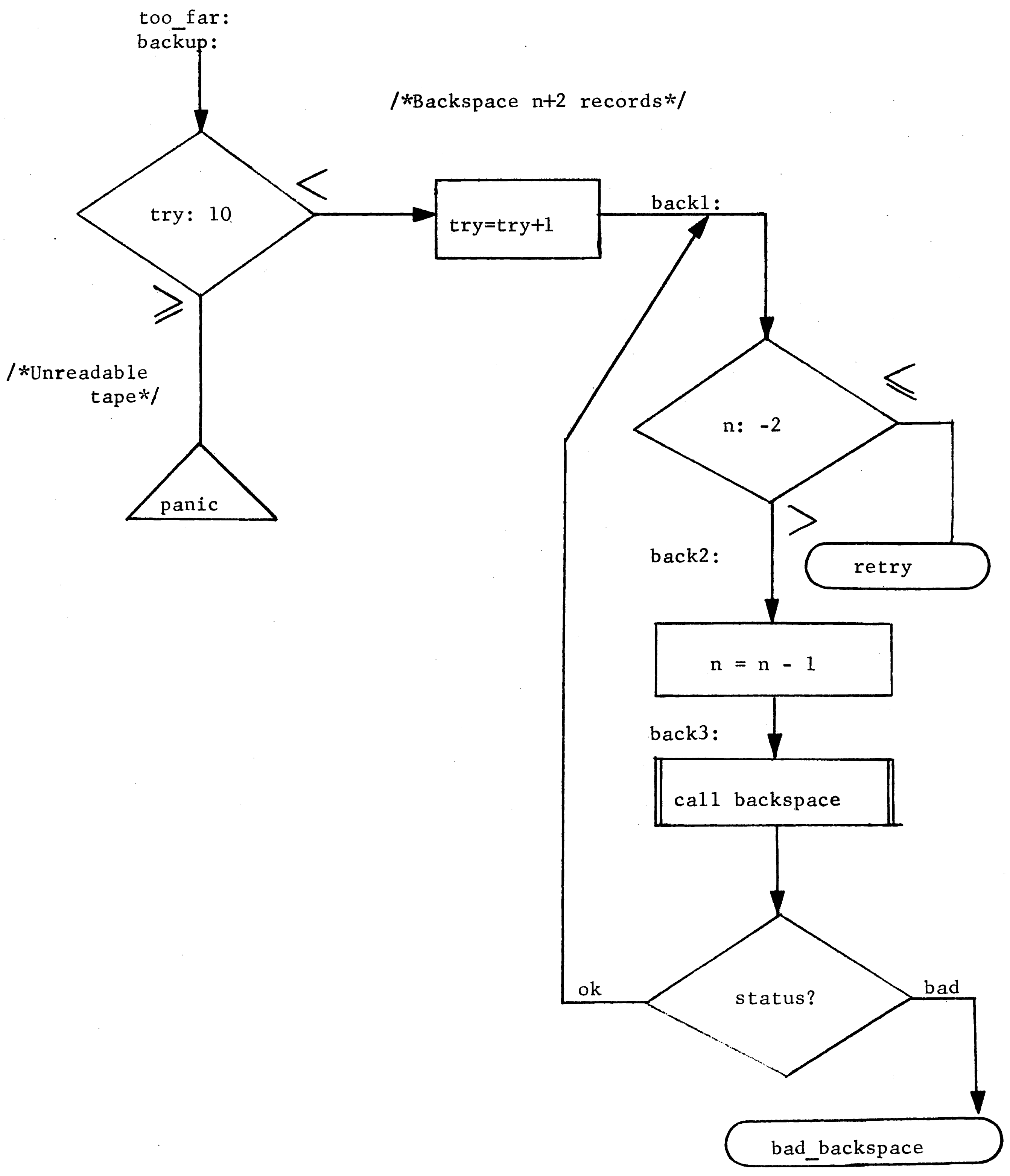
Continuation of Figure 3



Continuation of Figure 3



continuation of Figure 3



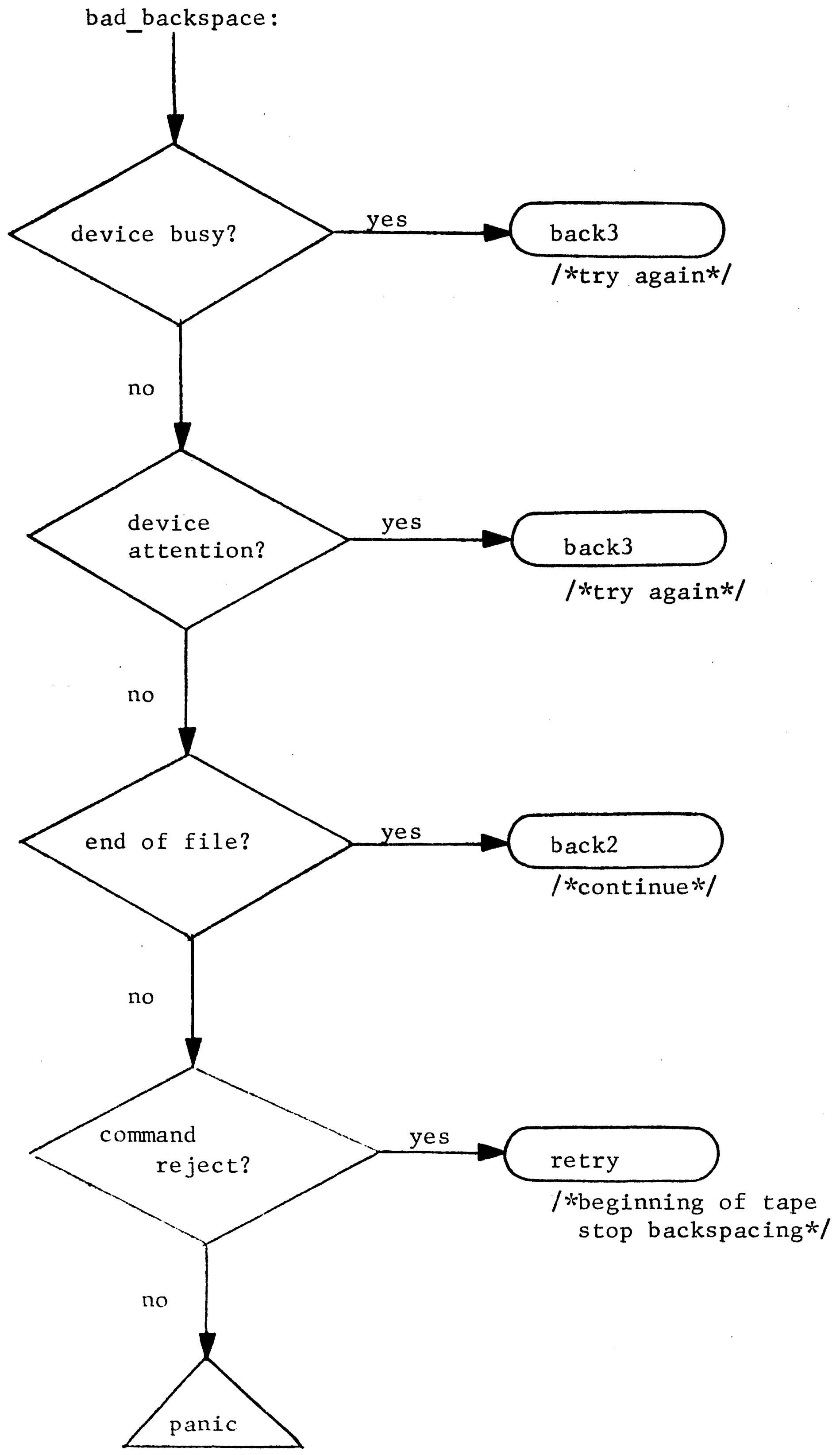


Figure 3 concluded