TO: MSPM Distribution
FROM: J. M. Grochow
SUBJECT: BD.9.04
DATE: 12/15/67

This revision of BD.9.04 reflects the changes described in BD.9.04A (8/4/67). The following additional changes are also included:

1. The final argument of condition and reversion has been deleted.

2. Some changes have been made in the management of default condition handlers.

3. The condition-handling primitives now reside in ring 1 through ring 63.

4. Calling of condition handlers is now done by "helpers" in the ring in which the handler was established in order to avoid protection violations.

Published: 12/15/67
(Supersedes: BD.9.04, 05/08/67)

Identification

Condition handling in Multics: condition, reversion, signal, set_default, find_condition.
R. M. Graham, M. A. Padlipsky, J. M. Grochow

Purpose

During the execution of a process in Multics, certain conditions may be encountered which necessitate action outside of the normal flow of control. That is, they must be dealt with whenever they happen to arise. In PL/I jargon, these conditions may be "on-conditions" or "interrupts"; in general Multics discourse, they are simply to be called "conditions" (usually prefaced with "system-defined" or "programmer-defined"), and are to be viewed as software analogues of hardware faults. Because individual programmers may define their own "conditions" and may signal the fact that these conditions have arisen, examples of conditions are difficult to give; they may range from the occurence of arithmetic overflow or a divide check (both system-defined) or something like "X_007" (clearly programmer-defined.) Indeed, despite the fact that some "conditions" do arise from actual hardware faults (overflow, e.g.), the fault-handling mechanism has been subsumed under the condition-handling mechanism in Multics. The role of the Fault Interceptor (BK.3) may be looked upon as the turning of a hardware fault into a condition "signal" as soon as possible. This view of fault-handling greatly facilitates the implementation of both user-provided fault handling for non-reserved faults and protection of system-required fault-handling for reserved faults.

Conditions of interest may, of course, arise at any time during the execution of a process. (This is particularly true of error conditions, which as a matter of Multics policy are to be treated by means of condition signals and condition handlers; see BY.11.) Also, specification of handlers for conditions may be changed at almost any time during the execution of a process. This unpredictability of timing means that there is no a priori way of determining which protection ring control will be in when a "condition" arises; therefore, condition handling must be under the aegis of the protection mechanism. The present section, then, describes the mechanism whereby Multics permits the controlled transmission of and response to condition signals. Control is of paramount importance, because in the abstract signals are in some sense independent

of protection rings and unless constrained could lead to unauthorized wall-crossings. The primitives described herein remove conditions and their signaling from the realm of protection-independence and fit them into the fabric of the overall Multics protection mechanism.

It should be noted that, in the interests of clarity, this section alters somewhat the PL/I terminology in the area of "conditions": We shall speak of establishing a condition handler, or perhaps of defining a condition, and of signaling a condition, or simply signaling. As condition-handlers for a given condition may supplant one another dynamically, we shall refer to that handler which control is intended to be passed to at a given point in time as the "active" or "most recently established" handler. When it is appropriate to distinguish between kinds of conditions, the aforementioned "system-defined" and "programmer-defined" prefacing will be employed.

Introduction

Three subroutines comprise the system-wide, primitive condition-handling machinery of Multics: condition, reversion, and signal. The condition routine is invoked to establish a condition name and its handler; after a call to condition (until further notice) the specified handler may be thought of as "active", and in the absence of protection constraints control will pass to it if and when the named condition is signalled. The reversion routine serves as the "further notice" alluded to in the preceding sentence: it causes the currently-active handler for a named condition to be replaced by the handler which was most recently previously active (but, for any "condition", it will not cause a reversion beyond the default handler; this point will be expanded upon below). The signal routine is invoked to signal the occurance of a named condition; when it is invoked control passes to the currently-active condition handler through the ring in which it was established. Programmers - particularly system programmers - may invoke these routines directly; on the other hand, compilers may also compile direct calls to them as appropriate. The routines are, then, in some sense "system" primitives.

Restriction

In Initial Multics, signalling will not be performed in ring 0.

Overview

Continuing the analogy between condition handling and fault handling, we find that the central device in the condition handling scheme about to be described is a "signal vector" - a software analogue of the 645 hardware's "fault

vector". For each process, the signal vector is implemented as a series of segments, one segment per protection ring, which contain threaded push-down lists of condition handlers, one list per condition. For a ring numbered n, the signal vector is contained in a segment named <signals_n>. As is the case with the call-save-return stack for each ring (see BD.9.01), the illusion is preserved for the user that there is but one signal vector. Details of the construction of the signal vector are discussed under Implementation, below. The main things to note at the overview level are that each condition, system-defined and programmer-defined, is dealt with by placing its handler on top of its corresponding list in the signal vector for the ring in which the condition is established (at the point in the process when the condition is established and the desired handler is specified, that is), and that the maintenance of the signal vector(s) is carried out such that when a condition is signalled the proper, active handler will be invoked regardless of intervening protection wall crossings. As will be seen, the method of accessing signal vectors also provides the capability of constraining the handling of selected conditions to be performed by the corresponding condition handlers in specified protection rings. (The signal vector also contains a special push-down list for the Unwinder - see BD.9.05 - under the "condition name" cleanup, which name is reserved.)

The pushing and popping of the list of handlers for a given condition is performed by two of the three subroutines which comprise the condition-handling mechanism. To push down the list for condition condname and specify that procedure proc is the active handler for it:

```
call condition ("condname", proc);
```

By the nature of push-down lists, a subsequent invocation of condition for condname in a process will of course establish whatever procedure - say proc_2 - is specified therein as the active handler. Suppose that such action has been taken and then at some later point in the process it is no longer desired to have condname handled by proc_2, but to revert to the immediately previous handler: to pop up the list of handlers for condname

```
call reversion ("condname");
```

After this call, in our little example, proc is again the active handler and if condname is signalled proc will be invoked.

Note that the names "condition" and "reversion" are chosen to suggest similarity to, but not necessarily identity with, the PL/I "on-condition" and "revert" statements. With these routines as primitives, PL/I - and, indeed, any other Multics language which contains similar facilities - will be able to implement its own interpretation of conditions within the framework of the Multics protection mechanism.

The task of investigating signal vectors to determine the identity and cause the invocation of the proper handler for a given condition falls to the third of the condition handling subroutines, signal. Following the notation and continuing the example of the preceding paragraph, then, to cause the active handler for condname to be invoked:

```
call signal("condname", flag, ptr)
```

(The name "cleanup" is explicitly barred from being an acceptable value for condname; see Implementation and BD.9.05. The additional arguments are explained in Implementation and are not of interest in the Overview.)

In the absence of any restrictions on the scope of condname, proc will be invoked; the call to proc, be it noted, is performed from the ring in which proc was established, going through the ring in which the signal was made. Thus there can be as many as two ring crossings involved; if a handler residing in ring n is put in the signal vector of ring m and the signal occurs in ring p, control first passes from ring p to ring m by a call to the helper in ring m, and then to ring n by the call from the helper to proc. The first ring crossing allows any argument supplied by the signalling procedure to be validated and the second ring crossing allows the Gatekeeper to check that the handler can be called from the ring in which it was established. (The implementation of signal, as will be seen, facilitates the calling of proc from the proper ring by invoking a ring-1 adjunct called signal_search with a formal ring-crossing; and then a helper in the proper ring to actually call the handler; this point is not of interest at the overview level, and signal will be treated as a single conceptual unit here.)

In order to present even an overview of the operation of signal, a further piece of apparatus must be introduced at this point: Associated with each signal vector is a linkage section (<signals_n.link>, for ring n). To search this linkage section, all of the condition handling subroutines rely on the generate_ptr library routine (see BY.13.02); this routine will return a pointer to what

is actually the head word of the (threaded) push-down list in <signals_n> for condname, as if condname were an external symbol (which it is). At the head of the list is a pointer to that member of the list which is currently at the top of the list (i.e., is the entry in the signal vector which would correspond to the active condition handler if there were really only one signal vector). What signal does, then, is first to search for condname in <signals_n.link>, where n is the ring from which it was invoked. If an entry is found, signal must determine whether this particular entry is actually the active one - for it is possible that the top of the list for ring n was established during a prior entry into the ring and that the actual most recent establishing of a handler for condname took place in another ring entirely, and hence is to be found in, say, <signals_m>, m ≠ n. The basis on which this determination is made is rather straightforward: whenever an entry is made in a signal vector (by condition) the then-current value of the Gatekeeper's "invocation number" (see also BD.9.01) is placed in the entry. The invocation number is in reality the index into the Gatekeeper's <rtn_stk> for the entry on that stack which contains the return information for a particular all; its current value is always placed in a fixed location (<stack_n>|2) when the Gatekeeper effects entry into a new protection ring for a call. Then if the invocation number in the first entry found in <signals_n> is the same as the now-current value (i.e., current at the calling of signal) the entry is the one we are looking for. If not, change the now-current value of the invocation number to the previous value, and investigate <signals_k>, where k is the ring number of the procedure which originally called into ring n, ring n being where the procedure which called signal is (k is obtainable from the Gatekeeper's <rtn_stk>). If the top-of-the-list entry for condname in <signals_k> has this new invocation number, then it is the active handler's entry, as it must have been established during the execution of a procedure which executed just prior to the procedure(s) in the ring from which signal was called, else its invocation number would not be the one previous to the calling of signal. And so on ... The extension of the searching process may or may not be obvious; at any rate, it is deferred until the discussion of the implementation of signal. Figure 1 offers a schematic view of inter-ring condition establishing and signaling.

Since each entry for a condition handler in a particular signals_n segment contains information pertaining to the procedure which caused the entry to be established, the user should be sure to call reversion before returning from a procedure for any condition names he has called condition for in the same procedure. Failure to do this may cause chaos during future attempts to "signal" these condition names.

Each "call" in a user's program that involves a ring crossing will cause information to be added to the <rtn_stk> and thus increment the invocation number. As returns are made this information is removed and the invocation number decremented. It is thus possible that condition handlers will be left in various signals_n segments with invocation numbers greater than the current invocation number (or in some cases with an invocation number equal to the present invocation number but with references to stack frames that are no longer active).

The procedure "signal_search" is invoked by signal to search signals_n segments in other than the current ring. Any signal vector entries that signal_search encounters with an invocation number greater than the invocation number current for the ring in which it is searching will be removed by a special call to reversion$ring with arguments condname (char(*)) and rnum (char(2)). A message will also be put in the user's error file to indicate the action taken. Note, however, that this correction of user mistakes (reverting condition handlers) only occurs when the mistake is encountered in searching.

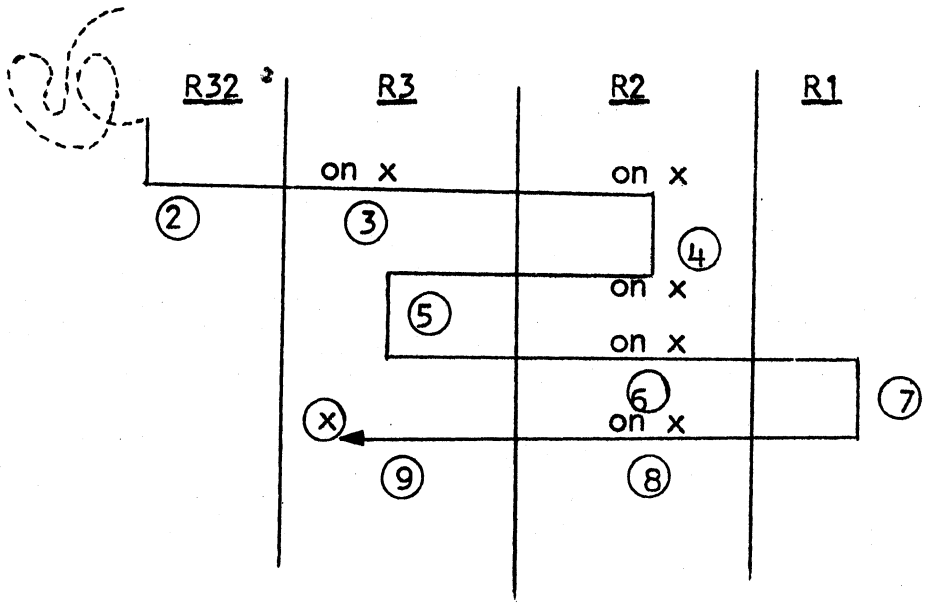
Another role the linkage section plays is found in the area of defining default condition handlers. For system-defined (reserved) conditions, it is possible to indicate "trap before definition" (see BD.7.01) in the condition's entry in the pre-initialized linkage section for a given ring (<signals_n.link> may, indeed, even contain different information than <signals_m.link>). When the linkage section is being examined to find the definition of a condition-name-as-external-symbol, the trap will be sprung and the subroutine to which the trap pointer points will be invoked. The subroutine will be privileged to write in <signals_n> and will start a push-down list for the condition in question, with the first entry (active until and unless condition is invoked) establishing the appropriate procedure as handler for the condition. The routine to do this is set_default (see below).

Once a default handler has been established, reversion will respect it, never popping a condition's push-down list of handlers past it. The default handling for programmer defined conditions is less elaborate: if no definition is found for condname in any signal vector's linkage section (in those rings involved in the pending return list of <rtn_stk>), signal starts a search for the current handler for condition "unclaimed_signal." (See BY.11.05).

The Multics Shell, for example, will in general respond to an unclaimed_signal condition by a call to print_err (as part of the standard system treatment of error-handling; see also section BY.11) and a return to command level. The "unclaimed_signal" condition is itself a system-defined condition, and if no handler is found for it, signal will execute a "terminate-process" fault (see BB.5.03).

The linkage section of the signal vector also affords the means of the constraining certain conditions to be handled only in certain rings. In the course of its search for the most recently established condition handler, signal investigates the "class code" in the linkage entries it encounters for condname (see Figure 2). Subsequent actions (detailed in the implementation description) are subject to the dictates of the particular code encountered. Some possible codes and their meanings are: 0, no constraints on this condition (that is, the search may enter and/or leave the protection ring in which the code is found); 11, may not leave ring, but may enter; 12, may not enter or leave ring, but may pass over; 13, may not enter, leave or pass over, 14, may not enter, but may leave. In the definitions, "enter" covers the cases when signal was invoked from ring n, has not found the active handler in <signals_n>, and is searching <signals_m.link> - or "entering" ring m, where m is the ring from which n was called (and to which it will return, of course, as indicated by <rtn_stk>); "leave" covers the cases where signal was invoked from ring n, has not found the active handler in <signals_n>, and is prepared to search <signals_m.link> - if it is permitted to do so by the code in <signals_n.link> (that is, if signal may not "leave" n, it will invoke the default handler in <signals_n> rather than search <signals_m.link>). It can be seen, then, that the handling of a condition can be controlled on a per-ring basis by suitable class coding in linkage section entries. System-defined conditions will usually be so constrained. Programmer-defined conditions may be so constrained as well.

Figure 1. Finding the "most recently established" condition handler



The wandering arrow represents control in a process, circled numbers are invocation numbers, "on x" indicates a call to condition for a condition named x. (The R_i are protection rings.)

When the call to signal for x is made (at ⓧ in the figure), <signals_3> will be investigated first. At the top of the x list is the handler specified in the call at ③, but its invocation number (③) is not the same as the current one (⑨), so the search continues - in this case to ring 2. At the top of the x list in <signals_2> is the handler specified in the call at ⑧; this is the active handler, as the "current" invocation number is indeed 8 (9 minus 1).

(Note that after return from ⑨ to R_2 a call to reversion would cause the handler specified at ⑥ to become active, and so on.)

Figure 2. <signals_n.link> entry

~~ indicates irrelevant to condition-handling

"value" is offset of head-word for condname in <signals_n>

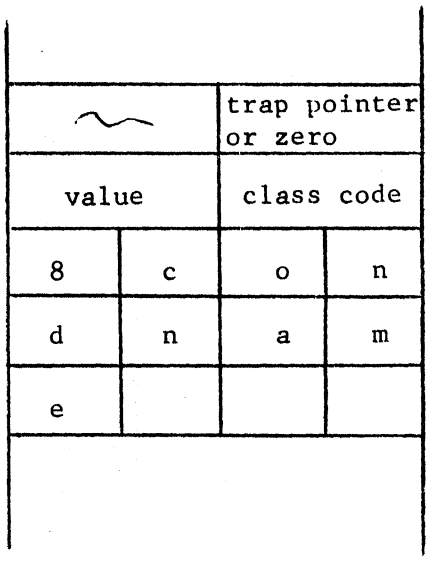
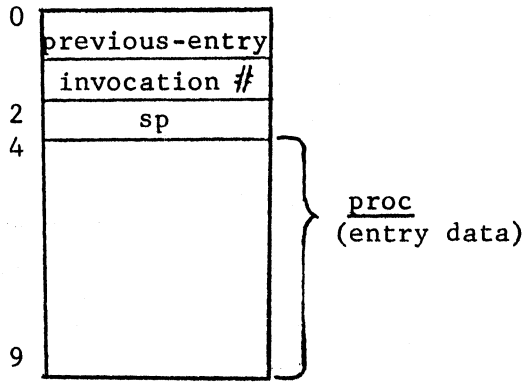


Figure 3. Signal Vector entry



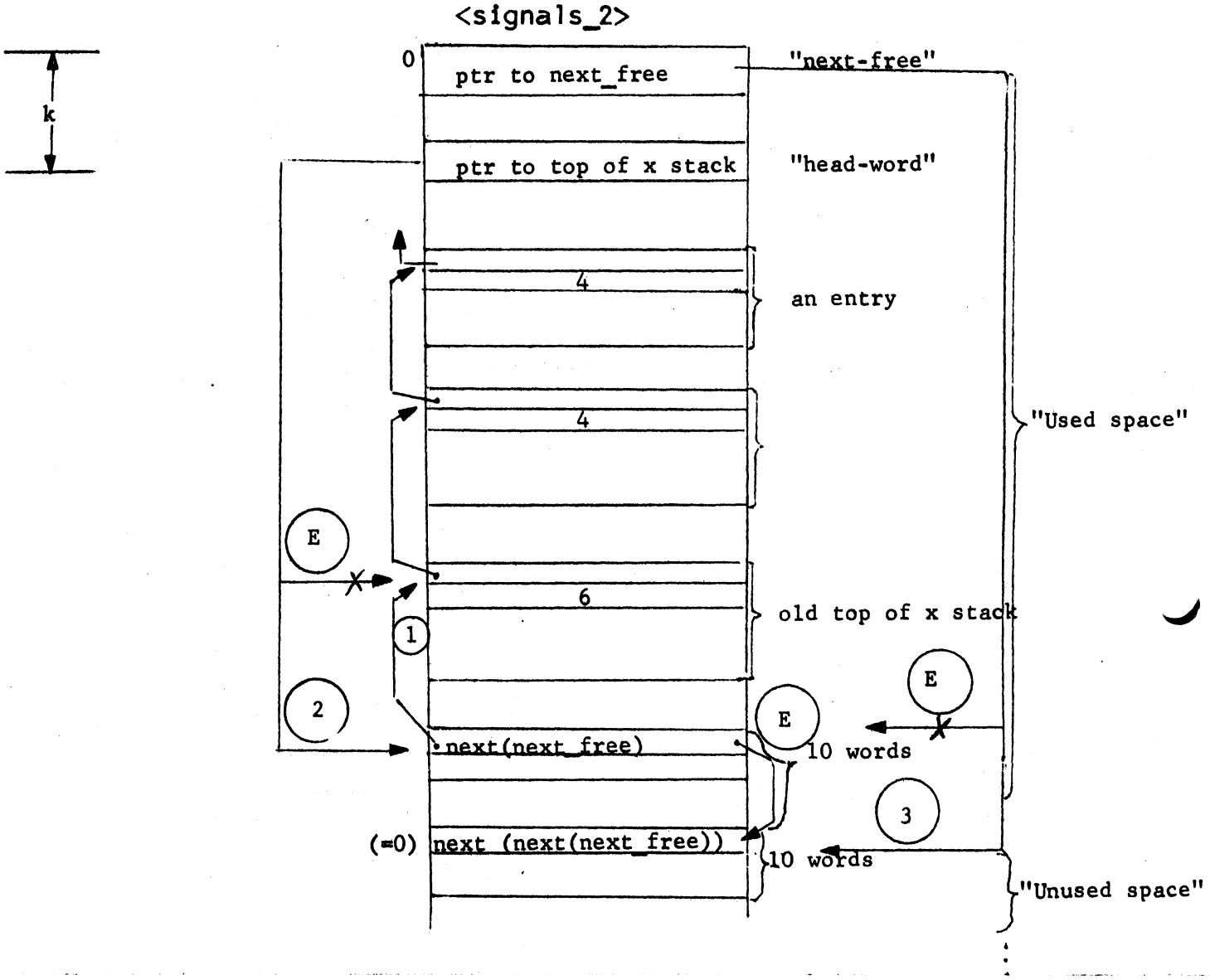
Notes:

1. Previous-entry: if zero and on some condition's push-down list, then is first entry; if 0 and on free block thread, then is in "unused" space (see also Figure 4, and Implementation).

2. Invocation_no: if a zero, indicates that this is a default handler entry.

3. sp: pointer to stack frame of procedure which invoked condition (contents of sp|16); for use of Unwinder (see BD.9.05).

Figure 4. A signal vector and a handler list within it



Notes

1. "k" corresponds to "value" in the linkage section entry (cf. Figure 2).
2. The numbers inside the various entries are invocation numbers corresponding to the calls to condition indicated in Figure 1, R2.
3. Pointers (arrows) marked E are established at the time of entry to condition to add a new handler.
4. Numbered pointers indicate order of setting by condition, X's indicate destruction by condition.
5. The "next_free" and "next( )" notation is explained under Implementation, passim.

Implementation

The remainder of this section deals with the implementation of the Multics condition-handling primitives. Recall that condition pushes down the list for a given condition's handlers in the signal vector, reversion pops up the list, and signal finds and invokes the most recently established ("active") handler.

Figure 3 shows the general form of an entry in a signal vector; Figure 4 presents a sample portion of a signal vector (actually, it corresponds to the situation diagrammed in Figure 1, and is <signals_2>.) The signal vector segment is tightly packed and multiply-threaded in the interests of space-saving and flexibility. The threading allows the several push-down lists in the segment to increase in length without the restrictions which would be imposed by a fixed-length block structuring and at the same time makes for compact segments when the lists are relatively short. Note that a relative pointer to the next free block of ten words in the segment is maintained in <signals_n>|0; if that block begins with a 0-word, it is in the unused space at the end of the segment; if it begins with a non-0 word, the word is a relative pointer to the next free block within the used portion. The segments <signals_n> and <signals_n.link> are preinitialized for system-defined conditions, and reside in the system library.

The details of the implementation are presented in the context of the accompanying block diagrams, which will repay close attention in certain of the more complicated cases.

CONDITION

The calling sequence is

```
call condition (condname, proc);
```

with declarations

```
dcl condname char (*), proc entry;
```

where condname and proc are as in the Overview.

Figure 5 presents a block diagram of condition. The notation of the Overview is continued below. The logic of condition is as follows:

1. Save ring. The ring number of the calling procedure is obtained from <process_info> (BD.6.11), call it n.

2. Does definition exist? Call generate_ptr (BX.13.02) with condname and "signals_n" as arguments. If condname has previously been defined, a pointer to its linkage section entry is returned; no new list is created and we proceed from step 4. However, if condname has not been defined (a null pointer is returned from generate_ptr), a new list must be started and we proceed as specified in step 3. (Note that the trap-before-definition attribute in the linkage section entry for a system-defined condition will be "sprung" by generate_ptr; therefore, the only case in which condition will receive no pointer from generate_ptr is that of the first reference to a programmer-defined condition).

3. Start list. An empty condition handler list is started as follows:

   a. Get the location of the next_free locations from <signals_n>|0 and update the free space thread.

   b. Call link_change$make_definition (segment, symbol, value, class); link_change$make_definition (see BY.13.03) builds a linkage segment entry with arguments segment= "signals_n", symbol= condname and value= the offset of condname within the segment <signals_n> (found in a. above); see also figure 2.

   c. Zero out the entire entry.

4. Add to list. (From either step 2 or step 3.) The logic for adding an entry to an established list is as follows:

   a. Get the next_free location pointer and update the free space thread (see steps 4-6 under reversion).

   b. Set previous_entry= the current value of head_word.

   c. Update the head_word to point to this entry.

   d. Invocation_no is found at sb|2.

e. sp is set to the stack pointer of the procedure that called condition. This is included so that the Unwinder will have this information if the condition involved is "cleanup".

5. Return

REVERSION

The calling sequence is

```
call reversion (condname);
```

with declaration

```
dcl condname char (*);
```

where condname has the same meaning as in condition.

Figure 6 presents a block diagram of reversion.

1. Save ring. The ring number of the calling procedure is obtained from <process_info> (BD.6.11), call it n. reversion$ring specifies n as an argument.

2. Obtain pointer to list. Call generate_ptr with condname and "signals_n" as arguments.

3. Return if no list or empty list. If the call to generate_pointer showed that the symbol condname was not defined returns to its caller. The head word for condname's list is located at <signals_n>|value where value comes from the linkage section entry (see also Figure 2). If the head_word=0 then the list is empty and reversion returns to its caller. If the invocation_no=0 then this is a default handler and reversion returns to its caller without reverting this handler. (Note that 0 can not be an actual invocation number since the smallest index into the <rtn_stk> is 2.)

4. Prepare to rethread. The current value of next_free must be saved (call it old_next_free), so that next_free can now be made to point to the entry about to be removed. (That is, old_next_free = next_free, and next_free = contents of head_word.)

5. Pop list. The active entry for condname is removed from the push-down list by the single expedient of setting its previous_entry value into head_word. This makes the previous entry on the list become the active one.

6. Rethread. The entry just removed is now a free block. To maintain continuity of free block threading, the first word (location previous_entry) must be set to the old_next_free value, which was saved in step 2.

7. Return

FIND_CONDITION

It is sometimes useful to be able to examine the contents of a handler list. The find_condition routine is furnished for this purpose; it operates in the protection ring it is invoked from.

The calling sequence is

```
call find_condition(condname, n, proc, flag);
```

with declarations

```
dcl condname char(*), (n, flag) fixed bin (17), proc label;
```

where

condname is as in condition

n is the number of places down in the list to examine (n=0 indicates the top of the list)

proc (returned by find_condition), is the handler indicated in the nth entry in the list or the last entry in the list if there are less than n+1 entries

flag is set to zero if proc is indeed the nth entry and is set to m, where m is the number of places down the list the entry is, if proc is the last entry rather than the nth. (If there is no list of handlers for condname in the current ring, flag is set to -1.)

SIGNAL

The calling sequence is

```
call signal (condname, rtn_flag, ptr);
```

with declarations

```
dcl condname char (*), rtn_flag fixed bin(17), ptr ptr;
```

where condname is as usual, rtn_flag is a switch indicating whether or not the caller will accept a return from the condition handler, and ptr will be passed to the condition handler as an argument, when signal invokes the handler.

Figure 7 presents a block diagram of signal. The implementation of signal is made less straightforward than that of condition and reversion because it must have access to the Gatekeeper's <rtn_stk> (see BD.9.01) and because of the necessity of reading and writing signal vector segments (<signals_n>) in inner rings when a call to signal is made from, say, a ring-32 procedure and <signals_32> does not contain the active handler for the condition in question. That is, condition and reversion may execute in whatever ring they are called from (i.e., they have ring brackets of 1,63,63 execute only; see BG.9.00 re ring brackets) because they will only need to access the signal vector in that ring; but signal, on the other hand, must be able to execute with full ring-1 privileges. It is not desirable, however, to make signal strictly a ring-1 procedure, because the likelihood is great that the active handler will be in the ring it was called from and introducing unnecessary protection wall crossings is inefficient. Therefore, the tasks assigned to the conceptual unit "signal" in the Overview are actually distributed among two routines: signal is the routine called by the user, with the same protection list as condition and reversion; it determines whether or not the active handler is in the ring it was invoked from, and, if not, invokes the second routine, signal_search, which requires a ring-crossing (ring brackets: 1,1,63). Then signal_search, able to be fully privileged as a ring-1 routine, will find the active handler in whatever ring it's in and return the handler's entry data (proc, as above) to signal. The call to proc is then made by invoking signal_helper_n (n is the ring of proc) which calls proc with the associated wall crossings.

1. Get ring number. The ring number of the invoking procedure is available from <process-info>(BD.6.11). Call it n.

2. Get invocation number. The current invocation number is available from sb|2 (that is, <stack_n>|2). Call it curinv. (Recall that the invocation number is the index into the Gatekeeper's <rtn_stk> for the entry corresponding to the return information which was stored when a procedure in the ring of the <stack_n> was called.)

3. Call generate_ptr. In like manner to condition and reversion, signal must call generate_ptr in order to determine the location of the head word of the list of handlers for condname. Call the pointer returned pointer and the class code returned class.

4. Is pointer null? If pointer is non-null, proceed to step 5. If pointer is null, condname has not been established for ring n; therefore, call signal_search. Note that pointer cannot be null for system-defined conditions because they are, by definition, defined as symbols marked trap before definition in <signals_n.link> and generate_ptr will have "sprung" the trap, thus invoking a procedure which establishes the default handler via an appropriate call to set_default. (If it turns out that signal has been invoked for a condname which was not established in any ring, or which has no active handler in any ring, signal_search will return to signal with no proc to invoke, and signal will initiate a search for the current handler for "unclaimed_signal." On return from signal_search, proceed to step 9., which corresponds to "out" in Figure 7.

5. Is handler active? Using pointer to find the headword for condname, and the headword to find the entry on the top of the push-down list of handlers for condname in <signals_n>, get the invocation number of the entry. (Recall that the invocation number of the entry was set from the value current when the entry was made.) If this value equals curinv, we have found the active (most recently established) handler; proceed to step 6. Otherwise, proceed from step 7.

6. Found. If the proc in pointer's top-of-the-list entry is null, proceed to step 10. Otherwise, call proc with the single argument ptr. In the event of a return from proc, proceed to step 10.

7. Not found. Three of the class codes which might have been found in the linkage section entry for condname prohibit searching beyond the current ring; therefore, if class equals 11, 12, or 13 it is necessary to employ the default handler in the current ring; proceed to step 8., which corresponds to "def" in Figure 7. Otherwise, the search may continue; invoke signal_search and on return proceed to step 9., which corresponds to "out" in the figure.

8. Default. If pointer's entry has a 0 invocation number it is the default handler's entry: call the proc of the entry with ptr as argument and proceed to step 10. if proc returns.

Otherwise, investigate the entry pointed to by the back pointer, determine if its back pointer is null, and so on until the default handler's entry is found. (Note that if a default handler is not found - probably the case for user defined conditions - signal initiates a search for "unclaimed_signal" - step 11.)

9. Out. (This step is taken on return from signal_search after step 4. or step 7.)

a. Search found a null handler. It is possible to specify a null proc in a condition handler list entry. If one was returned by signal_search (i.e., if the active handler is null) go to step 10.

b. Search succeeded. If signal_search returned a non-null proc, call signal_helper_n with arguments proc and ptr. signal_helper_n then calls proc with the single argument ptr. If this call is returned from, go to step 10.

c. Search failed. If signal_search returned a special null label for proc, this implies that the search failed to locate an active handler for condname; in this case, the default procedure in ring n must be found and invoked; proceed to step 8.

10. Return allowed? If rtn_flag = 1, the caller of signal is prepared to accept returns and control is returned to the caller. If rtn_flag = 1, the caller is not prepared to accept returns. In this case signal causes a terminate process fault.

11. No handler. condname is set equal to "unclaimed_signal" and control returns to step 3. An entry is made in the user's error file to indicate the action (see seterr, BY.11.01). A flag is also set so that if no handler is found for "unclaimed_signal" a "terminate-process" fault (see BB.5.03) is generated.

SIGNAL_SEARCH

The calling sequence is

```
n = signal_search (condname, proc);
```

with declarations

```
dcl condname char (*), proc entry, n char(2);
```

where condname is an input argument, proc is an output argument, and n is an output argument specifying the signal vector in which proc was found.

Figure 8 presents a block diagram of signal_search, which routine is intended to be called only by signal. Its role is to carry out the search for the active handler for condname, after signal has determined that the active handler is not in <signals_n>, where n is the ring in which signal was invoked.

The logic of signal_search is as follows:

1. Get invocation number. Because a formal ring-crossing may or may not have taken place on the call to signal_search from signal, it is necessary to exercise caution in determining where to start looking in the <rtn_stk> for the ring number of the procedure which called into the ring from which the call to signal came. That is, if signal was not called from ring 1 (and the likeliest case is that it was not), the top entry on the <rtn_stk> represents the return information for the return to the ring signal was operating in; in this case, the signal vector of the ring involved has already been searched and the entry containing it is in some sense superfluous to signal_search. Now, the invocation number stored at sb|2 (which is to say <stack_1>|2) represents the index into <rtn_stk> for the most recent entry, and must, therefore, be "decremented" (replaced by the previous invocation number) before we start to search. This number will still lead to the wrong entry unless there was no ring-crossing involved in the call to signal_search. What signal_search does, then, is determine whether it was called from ring 1 or not in the following fashion: the Gatekeeper will have set a cross-ring flag in the Stack frame prior to the one in which signal_search is operating if a ring-crossing had taken place in getting to signal_search (the frame is the "freak link" discussed in BD.9.01), so signal_search inspects the contents of the sixteenth word of the Stack frame pointed to by the current contents of sp|16 and if the cross-ring flag is set proceeds to step 2a, where the invocation number will be decremented. If the cross-ring flag is not set, skip the decrementing and proceed to step 2b. In either case, call the contents of sb|2 (the invocation number's fixed location) curinv.

2. Loop. (This step is taken from several points, including step 1. It controls the tracking-through of the signal vectors indicated by the ring numbers in <rtn_stk>; that is, the currently-unsatisfied returns' rings.)

a. Decrement curinv. On entry, decrementing curinv causes the routine to deal with the return information for the ring which called the ring containing signal (i.e., the top of the <rtn_stk> push-down list prior to the "superfluous" entry for the call to signal_search in ring 1). On loops, decrementing curinv causes the routine to deal with the return information for successively prior ring-crossings. See also Figure 1; the decrementing of curinv carries the search "back along the arrow".

b. Test curinv. If curinv has been decremented to 0, the <rtn_stk> has been completely investigated, as its first entry has a value of 1. Therefore, the search for the active handler has failed; proc is set to a special null value which signal will recognize and the routine returns ("out" in the figure). Otherwise, the search continues, per step 3.

3. Get new ring number. Extract the ring number from the <rtn_stk> entry corresponding to curinv. Call it n.

4. Call generate ptr. As the other condition-handling routines do, signal_search also makes use of generate_ptr. Again, call the pointer returned pointer and the class code class.

5. Test pointer. If pointer is null, there is not even a default handler for condname in the signal vector for the current ring; therefore, proceed to "loop" (step 2.) to try the next ring. If pointer is non-null proceed to step 6.

6. Check entry codes.

a. Code 12. If the class code for condname in the linkage section of the current ring's signal vector is 12, go to "loop" (step 2.). Entry to the current ring for a signal of condname is prohibited by code 12, but the search is permitted to continue.

b. Codes 13 and 14. If class equals either 13 or 14, proceed to "out" (step 2b.) Entry to the current ring for a signal of condname is prohibited by these codes, and the search is not permitted to continue.

7. Is handler active? If class passes the checks at step 6., the next step is to determine whether the active handler has been found. This will be the case if the invocation number in the signal vector entry found through pointer is equal to curinv (see also signal, step 5.) If so, set proc equal to the entry data found in the entry and return to caller; if the invocation number is greater than curinv call reversion$ring (condname) to get rid of this "left over" condition handler - go to step 4; if curinv is greater than the invocation number, go to step 8.

8. Check exit code. If class equals 11, the search may not leave the current ring; we must determine the default handler in the current ring, in like manner to step 7. of signal, and pass it back to signal. If class is not equal to 11 at this point, the search may continue to "loop" (step 2.).

If no default handler is found, set a flag and return so that signal may search for handlers for "unclaimed_signal."

SET_DEFAULT

Calling sequence is

```
call set_default (condname, proc)
```

with arguments as in condition. set_default is identical to condition except that:

a. The invocation_no in the entry is set to 0.
b. The sp in the entry is set to null.

Figure 5.

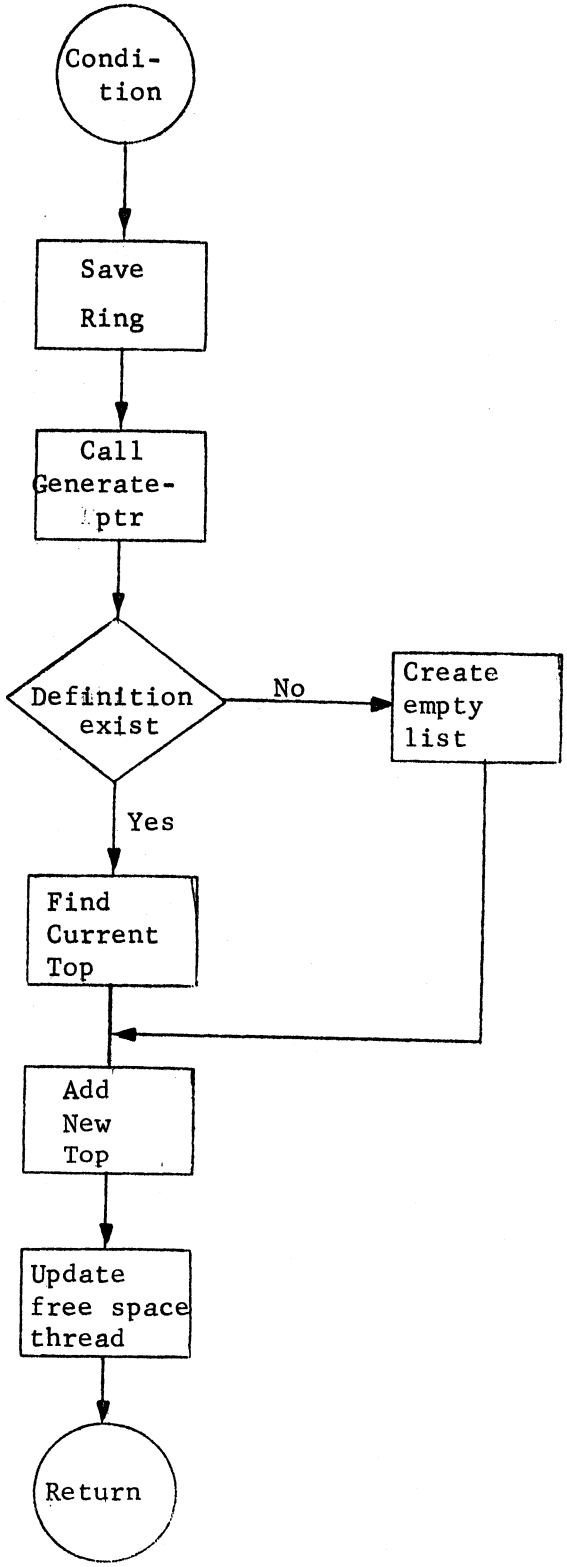

Condi-
tion
Save
Ring
Call
Generate-
ptr
Definition
exist
No
Create
empty
list
Yes
Find
Current
Top
Add
New
Top
Update
free space
thread
Return

Figure 6

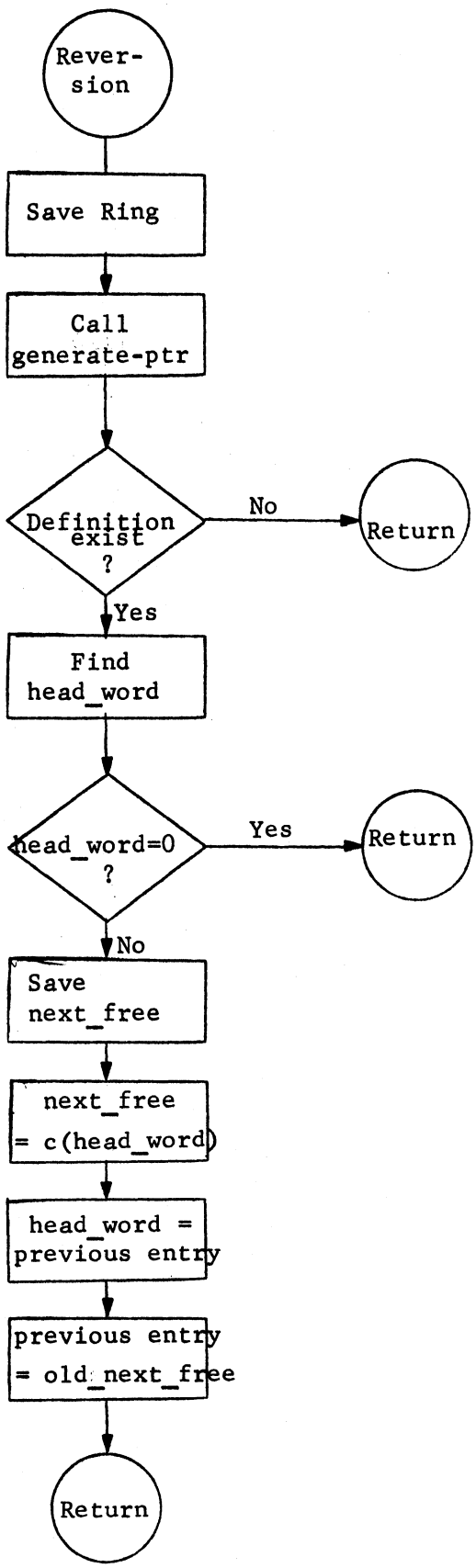
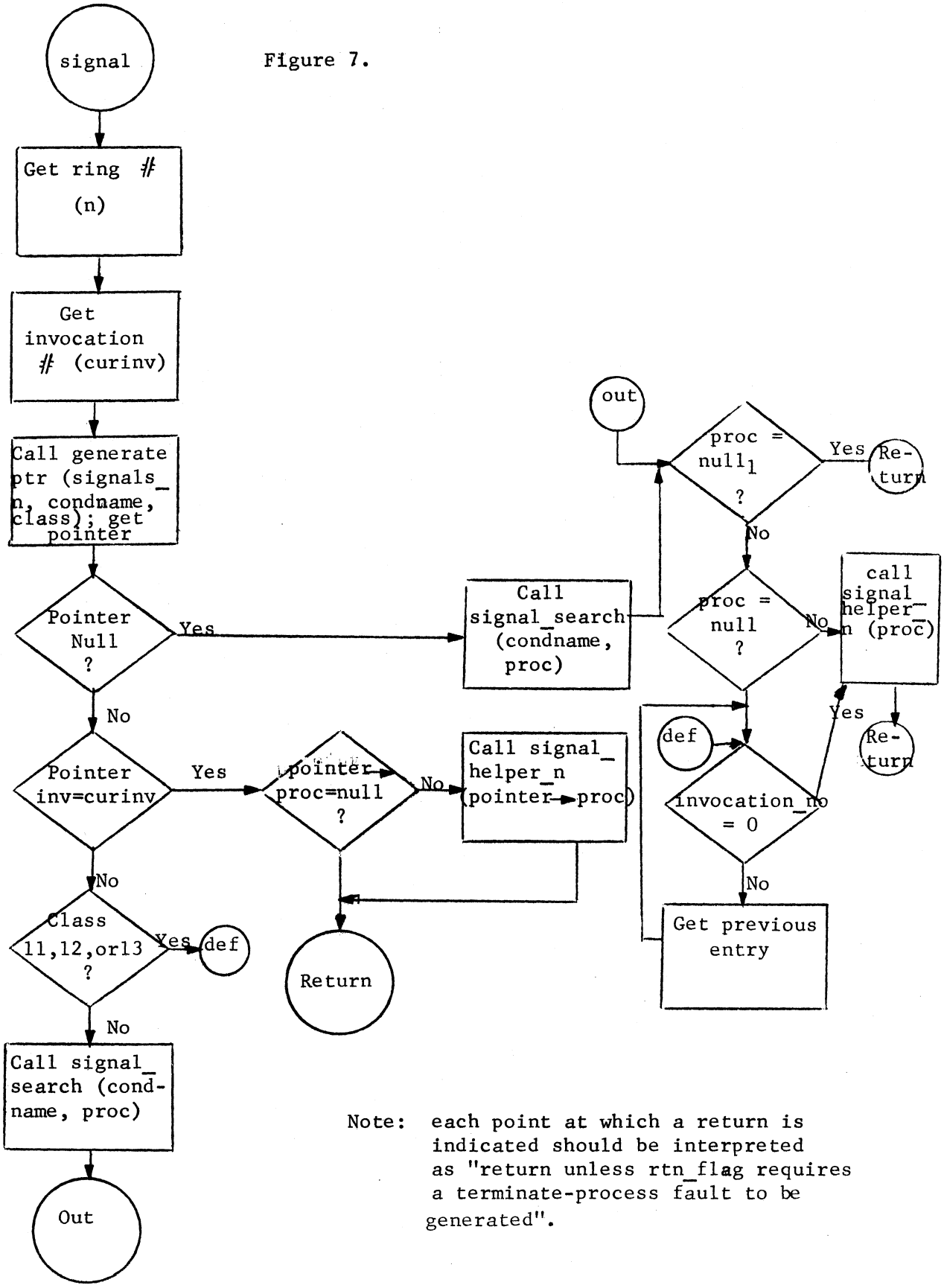
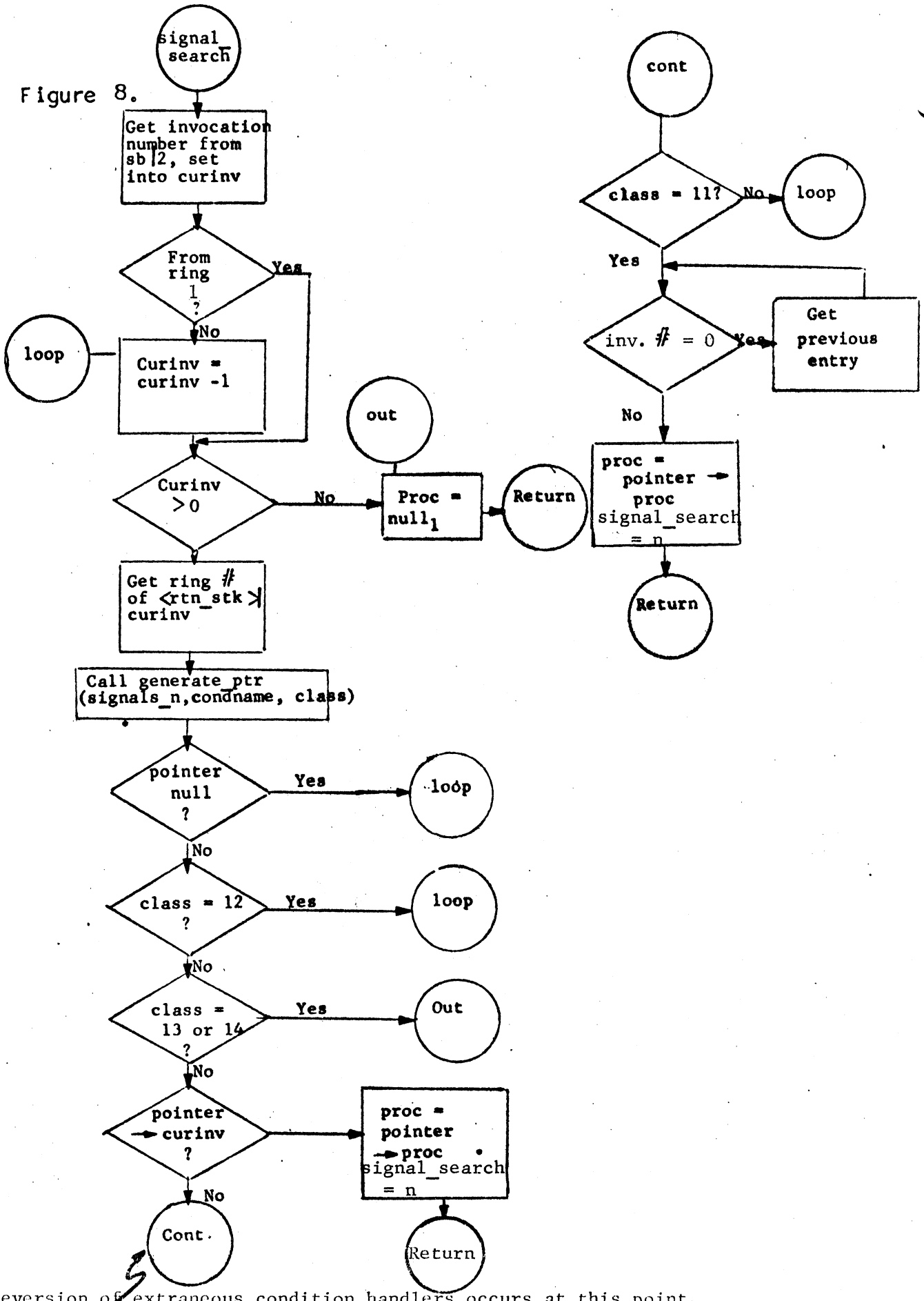

Rever-
sion
Save Ring
Call
generate-ptr
Definition
exist
?
No
Return
Yes
Find
head_word
head_word=0
?
Yes
Return
No
Save
next_free
next_free
= c(head_word)
head_word =
previous entry
previous entry
= old_next_free
Return

Figure 7.



Note: each point at which a return is indicated should be interpreted as "return unless rtn_flag requires a terminate-process fault to be generated".

Figure 8.



Reversion of extraneous condition handlers occurs at this point.