M0107

A GUIDE TO MULTICS

FOR

SUBSYSTEM WRITERS

CHAPTER V

Condition Handling and Abnormal Returns

Elliott I. Organick

Draft No. 3

February, 1969

Project MAC

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER V

CONDITION HANDLING AND ABNORMAL RETURNS

5.1 INTRODUCTION

Besides ordinary calls and returns, there are two other types of interprocedure (and possibly inter-ring) transfers which the user may wish to make and for which Multics lends system support:

(1) To execute a "signalled condition".

(2) To execute an abnormal return.

In this section we shall introduce the problems and issues involved in the implementation of each of these. With the motivation hopefully provided here, the reader may wish to read either or both of the remaining two sections of this chapter to see the details. As implemented in Multics, abnormal returns are executed with the aid of the condition handling mechanisms, so Section 5.3 cannot be effectively read independently of Section 5.2. "Condition handling" is a technical term in programming that is now well recognized as a result of the widely published specifications for PL/I. The term refers to an activity in which the user names in his program a hardware or a software condition and either explicitly or implicitly identifies (or supplies) code to be executed when the stated condition is detected at some later point in time (i.e., during execution of subsequent program steps). The remainder of this section discusses the pertinent PL/I concepts (and language specifications) that deal with condition handling. Persons already familiar with this aspect of PL/I may skip over this material.

In every programming system environment there exists a priori a class of system-defined conditions which can arise during execution that will fault the process. Some of these conditions are recognized (detected) by the hardware, while others are recognized during execution of system-supplied software. Examples of occurrences in this class might be accumulator overflow, zero divide, exceeding a subscript range, and attempts to perform illegal type conversions. Conditions like these are of the sort that are hardly ever completely avoided and hence are in the category of always-possible-though-always-unexpected. In PL/I, syntax is specified to handle occurrences of these conditions, providing the programmer a measure of choice of action and hence control over his program's fate. At the same time the machinery is universal enough in structure so that a uniform approach is possible for the handling of a wide range of condition types.

A set of a dozen or so "built-in", or system-defined conditions is enumerated in the PL/I specifications.* With each of these conditions there is associated a standard system action. This "standard" action is executed only in the default case, i.e., in the event the PL/I programmer fails to supply any other code for execution when a particular condition has been detected.

Subsystem designers are expected to recognize conditions which are not on the "built-in" list. Hence, additional machinery is provided so that the PL/I programmer can name other conditions and of course specify the actions which are to be taken when these conditions occur. Unlike the first category of system-defined or "built-in" conditions, this new category of programmer-defined conditions will not be automatically detected. Consequently, PL/I provides the subsystem programmer with the linguistic constructs which allow his subsystem to behave as a condition detector as well.

More specifically, three types of statements have been provided in PL/I:

	Purpose
ON statements	To designate a condition and the associated code which is to be executed when that condition is detected.
REVERT statements	To undo the effect of a previously executed ON statement that refers to the same condition that is named in the REVERT statement.
SIGNAL statements	To indicate occurrence of a built-in or programmer-defined condition.

A more complete explanation of these statements follows:

ON Statements

These identify a system-or programmer-defined condition and designate the corresponding code which is to be executed whenever that condition is detected. The general syntactical form of this statement is:

*For a complete list of these see Appendix 3, "IBM System 360 Operating System, PL/I Language Specification", Form C28-6571.

ON <designation of the condition*> <action specification>

where <action specification> may be null, a simple PL/I statement, or a block of code.

For example:

```
ON OVERFLOW BEGIN;
                    DECLARE SUM STATIC INITIAL (0) ;
                    SUM = SUM + 1;
                    IF SUM > 100 THEN
                    CALL OVERR;
                END;
```

designation of the condition

action specification

The action specification may be thought of as a "handler" for the specified condition. This is because execution of an ON statement has the effect of setting up its action specification as a body of code to be invoked later, as if it were invoked as a procedure. The execution of an ON statement can be said to establish a handler for the named condition. Execution of a subsequent statement that results in the detection of the named condition will cause an interruption of the main program sequence and the invocation of the established handler.

A natural question to ask is: What is the program scope in which a given established handler is said to be active i.e., how long does an established handler remain in effect? The answer is — up to but not beyond the point in time where the thread of control exits normally from the block in which the handler was established. This rule means that after executing an ON statement establishing a handler (which we will refer to as) Y for a condition named "X", the thread of control may pass through numerous other procedures (as a result of CALL or GO TO statements) before exiting from block B. All this time the handler Y would remain in effect. A handler is said to remain in effect or govern while executing in all the "dynamic descendents" of the block in which it (the handler) was established.

---

*The condition is designated by its name and, if programmer-defined, is indicated as such syntactically by placing the name in parentheses and prefixing it with the word CONDITION. Thus,

```
ON CONDITION(UNEXPECTED_DELAY) CALL PROCA
```

designation of condition — action spec.

There are two ways, however, to over-rule the effect of a handler Y. One may either temporarily replace Y with alternative code, say YY, or one may instantly nullify Y. A handler Y may be temporarily replaced if another ON statement for the same condition is interposed in the execution. If the second ON statement designates some other code (which we shall call) YY as the handler for "X", then YY is established and will remain in effect until YY's scope termination is reached. Termination occurs when a normal exit is taken from the block in which the handler was established.* The implied effect of "adding on" one handler after another for the same condition and in the same dynamic sequence, amounts to "stacking" handlers in a last-in, first-out discipline. To simply nullify the "rule" of a currently-effective handler, Y, one uses the REVERT statement.

The REVERT Statement

This names a condition (system- or programmer-defined) whose currently governing handler is to be nullified (i.e., popped from the current stack of handlers for "X").

```
REVERT X;
```

exemplifies the simple syntax of the REVERT statement. After executing such a statement, the previously established handler (or a system-defined default handler in case there are no more left on the stack) will be invoked if there is a subsequent detection of condition "X".

The SIGNAL Statement

This allows a programmer to indicate the occurrence of a condition that is named in this statement and thereby to cause its governing handler to be invoked. After execution of the indirectly designated handler, control will (normally) return to the statement immediately following the SIGNAL statement. It should be noted that the ON statement which established the currently effective handler for a condition "X" need not, and normally would not, appear in the same procedure(s) that contains the invoking SIGNAL statement.

Although any system-defined condition may be "signalled" with this type of statement it should be emphasized that executing a SIGNAL statement is the only way a programmer-defined condition handler can ever be invoked. Of course,

*Notice that if both Y and YY are established in the same block, then the scope of both handlers will terminate simultaneously upon exit from the block.

executing a SIGNAL statement does not alter the scope of the invoked handler. That is, there is no restriction on the number of times a statement like:

SIGNAL X;

may be executed to invoke the current handler for "X".

Signalling via the SIGNAL statement offers the programmer an attractive way to invoke a subroutine without actually having to specify its name, letting its designation be determined dynamically, as determined by the ON statement most recently executed. Whether this technique is practical depends on the particular machinery that is developed for the implementation of the ON, REVERT, and SIGNAL statements. As we shall see when we consider in detail the machinery developed for this purpose in the Multics environment, the overhead is high. Signalling activities are normally too costly to use except for special situations. These might arise when a program's complexity is already so great that the introduction of additional machinery for explicit invocation of specified actions (handlers) would add disproportionately to the debugging problems of the programmer. To make our point somewhat more specific, we sketch the following example:

Let a procedure <a> call <b> after first establishing the handler Y for condition "X". Let us further suppose it is appropriate or convenient for the test of occurrence (detection) of "X" to be accomplished in <b>. It is then natural to let <b> invoke the handler for "X" simply by executing the statement

SIGNAL X;

Executing this statement may, however, prove to be relatively costly. (One measure of this cost is the execution time required for invoked system routines to locate the desired handler.) If this is the case, a less expensive way can usually be arranged, simply by adding an additional argument (error code) in the call from <a> to <b>. If this done, <b> can be coded to detect the condition, set the error code parameter appropriately, and return. It is then <a>'s responsibility, upon receiving a return from <b> to test error code (before doing anything else) and to invoke Y (e.g., by a subroutine call) before proceeding with with other taks. Note that <a> knows what handler to invoke even though <b> may not.

5.1.1 Signalling Conditions in a Multi-Ring Environment

With special system-provided procedures, Multics makes it easy to provide condition handling in any process, regardless of the coding language that is used. These system procedures are:

< condition >
< reversion >
< signal >

They can be used by anyone to accomplish what can be achieved in PL/I with ON, REVERT, and SIGNAL statements. (It is no coincidence that the EPL and PL/I compilers generate calls to < condition > , < reversion > and < signal > in translating such statements.)

If we are going to gain a more sophisticated view of the condition handling machinery in Multics (present and future), it will be necessary to consider how it couples or should couple with the ring structure. To elaborate on this thought, we will do well first to walk through some of the steps of condition handling in the context of the Multics ring structure. (For this purpose we need no longer assume that PL/I is the programming language being used.)

Let us suppose a procedure < P > is executing in ring e. This procedure may enable* a condition named "x" simply by calling < condition > and designating the name of the condition, "x", and a handler, call it < procl > as arguments. The handler is a block of code in the form of an internal or external procedure. Also, let the ring number of < procl > be h (for handler).

After enabling condition x in this way, < P >, or any of its dynamic descendants is free to signal condition x. The need to issue the signal may be recognized in two basically different ways. Hardware faults may induce this recognition. In this case, a fault interceptor module can issue the signal for condition x. Alternatively, simple tests of state variables may be programmed by the user such that affirmative results are tantamount to event recognition. In this case, the procedure then executing, which shall be called the signaller, can execute the call to < signal >, naming "x" as an argument.

Signaller may be written either by the subsystem designer as a utility routine or it may be written by an ordinary user. (It makes little difference.) We shall assume that < signaller > executes in ring s.

*Here we shall be using the phrase enabling a condition to mean what in PL/I terminology is expressed as establishing a condition. In PL/I there exists additional mechanisms to enable or to disable a previously established condition. We shall not be concerned with this extra level of control in condition handling. Hopefully, therefore, confusion may be avoided.

From the foregoing "exercise" we see that up to three different rings may be involved. These are e (for enabler), h (for the handler), and s (for the signaller). A basic question to be answered is: Should a condition enabled in ring e be "signallable" from s if s≠e? A no answer would be tantamount to making calls to < signal > and to < condition > within one ring independent of those in all other rings. A yes answer amounts to saying that a condition enabled in one ring may be signalled from any other ring. This in turn implies existence of a mechanism for remembering in what ring the corresponding condition was enabled. It also implies that the same handler < p > can behave differently, depending on the ring from which < p > has been invoked.*

Arguments may be offered for both approaches. The first approach (s=e only) has the advantage of simplicity in implementation. It implies a minimum of execution overhead in invoking the intended handler. But it also implies that a programmer must know the rings in which his procedures will be executing when calls to <condition> and to < signal > are made, (i.e., he must be conscious of at least some ring crossings). This requirement is not in complete harmony with one Multics objective for the ring structure, namely to provide a compartmentalization "service" that requires no direct programmer involvement.

The second approach (s possibly ≠ e), clearly implies the complement of the aforementioned advantage and disadvantage, that is, more expensive signalling, but freedom to remain oblivious to ring crossings. In addition, however, there is one more important advantage. It is as follows: By permitting s≠e, we assure for instance that conditions enabled while a process executes in a user ring may be signalled from a supervisory ring (and vice versa). This provision, for example, allows a supervisor, that has intervened as a result of a user-incurred fault, to signal a system-defined condition which has been enabled in a user ring. In short, the system must be able to find a handler (which itself may be in any ring) that is established by user or by supervisor. Ideally, this capability should be replicated for the case of user-developed subsystems having two or more rings. Thus, an outer ring of such a subsystem would likewise be

---

*To see why this is so, let the ring brackets of the handler < p > for condition "x" be (u, v, w). Now, any signaller from ring s that has ring access to < p > will be able to invoke it. From our study of Chapter 4 we know that < p > would then execute in a ring, r, that lies somewhere in closed interval (u, v), depending on s. Suppose the enabled condition "x" is signalled more than once, and from k different rings, say, $s_1, \ldots, s_k$ (k > 1). The corresponding rings, r_k, in which < p > will then execute, may not all be identical. (That is, a handler invoked from different rings, and having an access bracket (u, v) such that u < v, may execute in different rings.) Although we may not have mentioned this previously, it is true that for reasons of protection, a procedure < p > is supplied with a separate copy of its linkage segment, < p, link >, for every ring in which < p > executes and hence may behave differently when executed. This somewhat surprising fact and its interesting implications are dealt with in Chapter 6. There we will show that a snapped link, generated by the Linker for the same symbolic reference, may depend on the ring of the link-faulting procedure.

able to establish handlers that can be invoked (explicitly, in this case) when an inner ring procedure executes a call to <signal> . This is just another expression of the Multics design philosophy that the interface between user and supervisory procedures function in the same way and using the same conventions as for an interface between different user-written procedures. In the initial implementation of Multics, the former, less general approach is taken. The more general approach has been studied carefully, however. A well thought out scheme has been proposed for achieving the general signalling mechanism which includes simpler schemes as subsets. It is too early to say if the general case will finally be implemented, or how. The scheme is described in Section 5.2.2.*

Some interesting problems arise:

(a) What ring relationships between h (handler) and s (signaller) should govern on whether or not the signalling procedure should be given access to the designated handler, <procl> ? The answer is that the controls which permit the signaller to call the handler apply here. E.g., s must be less than or equal to the outer ring of the call bracket for <procl>, and if within <procl>'s call bracket the desired entry point must also be a gate.

(b) In any process the condition x may be enabled more than once before it is signalled. Each enabling of condition x, even if from the same ring, may designate a different handler. Moreover, the handler may possibly be located in different rings. In addition to the question raised in (a), we must now add the question: which of the handlers is the one which should be asked to respond to the signaller (i.e., to which handler do we want control transferred)? The one we want shall be referred to as the currently active handler. Ordinarily, the answer is: the one designated when x was last enabled. But, whichever is the active one, how does the supervisory system go about locating it? Is a stacking scheme used? (The answer is yes.) If one pictures that signals pertain only to conditions enabled in the same ring, then it is easy to visualize how one might implement all three of the primitives, <condition>, <reversion>, and <signal>. A call of the form:

```
call condition ("x", <proc> );
```

when executed in ring e, might cause a pointer (i.e., entry datum) to <proc> to be pushed onto the top of a stack named "x" for ring e. A call of the form:

```
call reversion ("x");
```

*A complete design has been given in BD.9.04. dated 12/15/67.

in the same ring, e, would then cause the topmost element to be popped from the same stack. Finally, a call of the form

call signal ("x");

also executed in ring e, would cause the issuance of a call to the procedure whose entry datum is the topmost element on stack "x" (of ring e).

But what of the more general case if it were implemented in Multics? Here, signalling is not restricted to the ring in which the matching condition handler was enabled. What selection or searching process would be used to locate the desired handler? Would the programmer also have the option to restrict the search for an enabled handler so that it may be invoked only when the condition is signalled from specified rings? (The answer is yes.) How about reversion? Will the poping that is performed remain limited to the stack for "x" in the ring of <reversion>'s caller? (The answer is yes.)

(c) What practice is followed for the case where a procedure signals a condition that has never been enabled, or if enabled, has since been fully disabled (reverted)? In this regard it's important to be aware of the two kinds of conditions recognized in Multics.

(1) system defined

(2) programmer defined

It must be arranged somehow that the system behaves as if every system-defined condition is always enabled, each with a system-defined "default handler", i.e., one which will be invoked in case the user fails to impose one or more handlers of his own. It is not at all obvious how transfers to these default handlers are always assured in default situations.

A somewhat different mechanism must be devised for guaranteeing default handling of programmer-defined conditions, in such a way that the user still has an opportunity to interact effectively with his process, a prime objective or interactive processing.

The purpose of Section 5.2 is to explain the Multics solutions to the problems just raised.

5.1.2 Abnormal Returns

If, instead of transferring back to the point of call in the calling procedure, one attempts to execute a return to any other point in that procedure or in any other previously called procedure, we refer to this as an abnormal return. The corresponding PL/I terminology is "non-local go to". Imagine, for instance, that <a> calls <b>

calls <c> . . . etc., calls <t>. In principle it is possible to pass a label argument, say lab, from <a>, via <b> via <c>, etc., to <t>. While executing, <t> can then return to lab in <a>. The fact that this is a commonplace facility in MAD and in FORTRAN IV may give you the impression that no problems are presented here. Nothing could be further from the truth. Severe problems can be encountered in the proper handling of these returns when executing programs written in a more comprehensive language like PL/I and/or in a multi-ring environment, as the following introductory discussion hopes to show.

In the course of returning abnormally to <a> there is a matter of resetting the stack pointer to the target procedure's stack frame and recovering all the saved register values and the (indicators). This can be done relatively easily if all procedures <a> through <t> in the chain are in the same ring.* Returning to the earlier stack frame of <a>, which implies resetting the stack pointer, has the effect of recovering the storage allocated in the stack for variables of type "automatic" used in <b>, <c>, etc.

Unfortunately, there are several remaining recovery problems:

(1) There is a storage management problem which arises whenever a procedure allocates temporary (e.g., automatic) storage space in segments other than a stack segment. Usually, a procedure that allocates such space should also free it before executing a normal return. However, if such a procedure is bypassed during an abnormal return, there may be no opportunity to execute the code that recovers this allocated space.

Even if the user makes no explicit effort to allocate temporary variables in this way, the supervisor, or the compiler he is using may do so. Two examples are:

(a) The Multics standard way for handling all automatic varying strings.

(b) The EPL way for handling arrays of automatic varying strings. In both cases space selected for such data is taken from a free storage segment called <free_>. † Resetting the stack pointer for the abnormal return will not of itself accomplish the recovery of space that was allocated for such variables.

(2) During execution of the intermediate procedures (<b>, <c>,..., <t>) various conditions may have been enabled. As we have already

---

*A "standard" abnormal return sequence can be devised and in fact was once proposed for use in these situations - but later discarded.

†The details can be found in BP.2.02 and BB.2 sections of the MSPM.

suggested, each enabling of a condition amounts to the stacking of a pointer to a desired procedure or "handler". These pointers would be popped off such stacks prior to executing a normal return. When an abnormal return to <a> is executed, unwanted pointers should be popped from whatever stacks they have been put on. If these pointers were held in the stack frames, reversion of these handlers might be automatic. In fact, however, the pointers are kept in a special segment or segments. An extra effort is therefore required in popping these entries from their respective stacks during abnormal returns. Details will be given in Section 5.3.

(3) All these problems are made more complicated when the procedures and the stacks that are involved in the chain we intend to bypass reside in different rings. To perform an abnormal return, it is really necessary in fact to march backward through the chain of about-to-be bypassed procedures, one-by-one. During this backward march we must perform on each procedure the necessary "cleanup" operations, i.e., returning allocated space to free storage and popping pointers to condition handlers. This slow retreat, called "unwinding", is in fact what must happen when a user wishes to make what, from his source level language, seems like a nifty "end-run".

Because of the decision to permit unwinding across rings, the unwinding process is not only slow, but, for protection reasons, cannot in general be entrusted to any but a ring 0 supervisory procedure. (Abnormal returns are never executed from ring 0, and user-written routines which execute abnormal returns cannot be allowed unsupervised freedom to bypass ring 0 routines.) A special system procedure called the Unwinder is therefore provided. This procedure interfaces with the Gatekeeper and with the condition handling procedures to carry out its task. Moreover, it is legislated that whenever a user wishes to perform an abnormal return he does so by a call to the Unwinder. In some situations, a user will invoke the Unwinder mechanism without being conscious of it. For example, compilers like the EPL compiler will generate calls to the Unwinder when translating non-local go to statements.

5.2 CONDITION HANDLING — DETAILS

The first part of this section reviews the plan for condition handling, roughly as it is now implemented in Multics. This is the scheme which limits signalling to the ring in which the intended handler has been enabled. The second part of this section considers the more general mechanisms which make signalling from other rings feasible. Some knowledge of the general mechanism is needed to appreciate the abnormal return discussions in Section 5.3.

Each process is provided a specially designed data base for use in condition handling. It consists of a series of segments, one for each ring r, other than zero, of the executing process (r = 1, 2, ··· , 63). To conform with the MSPM terminology in BD. 9. 04, we shall call these, <signals_r>* where r is a two-digit character string representation of the integers, i. e., "01", "02", etc.

Pointers to condition handlers are saved in the various <signals_r> segments in entries that are threaded as push down lists. Several lists may be kept in a single <signals_r> segment, one for each distinctly named condition.

5. 2. 1 Details of the Current Implementation

Each call to <condition> has the effect of stacking a pointer to a handler. If a procedure <p>, executing in ring s, calls <condition>, e. g.,

call condition ("condname", proc);

the effect is to stack a pointer to the handler (<proc>) for a condition named "condname", in <signals_s>. The pointer, which is a six-word entry datum together with certain other information form a stack entry that is threaded with other entries for conditions having the same name.

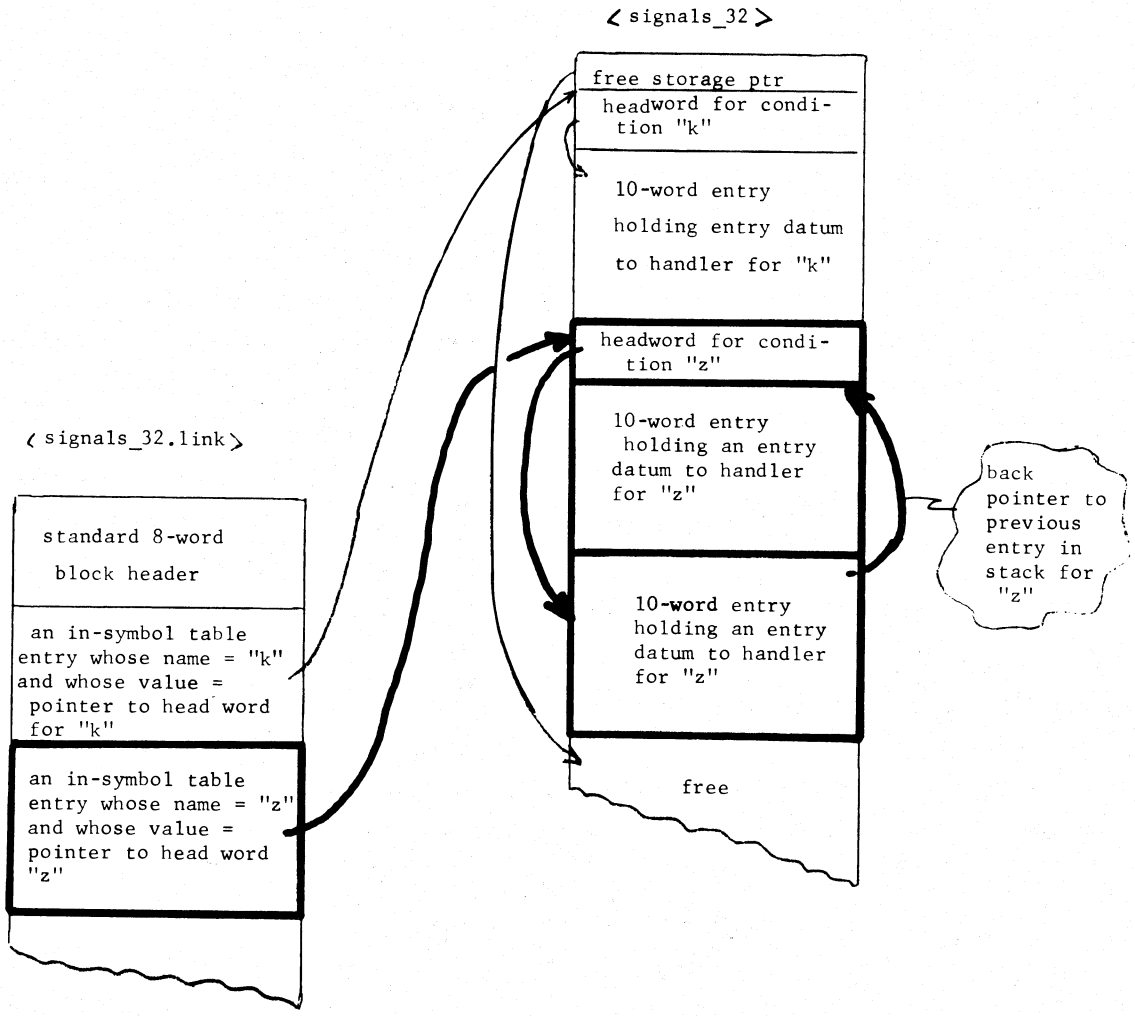
As a further aid in visualizing how stacking of entries for handlers are dealt with, Figure 5-1 presents a schematic view of the storage structure for <signals_32> and for its linkage segment based on the design given in BD. 9. 04 of the MSPM. †

By virtue of its special design, a signals segment can hold stacks for an arbitrary number of distinctly named conditions. Each stack consists of a "headword" and a threaded list of 10-word entries. The headword points to the most recently stacked entry. Each entry is back-threaded to its predecessor (if it has one). Space for the stacked entries is drawn as required from a free storage area within the signals segment.

---

*In the interim implementation of Multics that is in current use, there is actually only one ring outside ring 0, namely, ring 1. Hence, only one segment is involved as the data base for condition handling. It is called <cstk>. No MSPM documentation as yet describes <cstk>. Our approach in this discussion is to imagine the replication of <cstk> over the signalling domain that would include all rings 1 through 63, and to picture the data base as having the structure originally designed for it in BD. 9. 04.

†The initial implementation in actual fact uses a somewhat simpler storage structure in which the need for a linkage segment is eliminated. The structure presented in Figure 5-1 was chosen because it contains the same conceptual characteristics needed to illustrate the current implementation and also conforms with the design for the more general condition handling scheme which is outlined later in this section.



There are two conditions which have been enabled, "k" and "z". Two entries have been stacked for "z" and only one for "k".

Figure 5-1. Schematic of the Storage Structure for a Signals Segment and Its Companion Linkage Segment

The linkage segment contains in-symbol table entries, one for each named condition. Each entry in this table is basically a name-value pair. The value is interpreted as the offset in the corresponding signals segment of the headword for the stack that is associated with the name. The in-symbol table is searched on each call to <condition>, <reversion> or <signal> to locate the top most element of the appropriate stack.

Thinking in terms of ring 32, for instance, in-symbol table entries for system-defined conditions are always preset in <signals_32.link> by the system. Those for programmer-defined conditions are added as needed. That is, upon calling <condition> with a new name, a search of <signals_32.link> reveals a need to add a new entry. Each "first" call to <condition> also causes a new headword to be set up in <signals_32>. Of course, each call to condition, including first calls, also results in the addition of a 10-word entry to its appropriate stack.

5.2.1.1 Popping Handlers

Each call to <reversion> has the effect of popping the top "handler" from a given stack. For example, the procedure <p> in ring s may, just before returning to its caller, execute a call like

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

The effect would be to remove the top (most recently added) entry from the stack in <signals_s> that is associated with "condname". The system automatically supplies stack entries to "default handlers" for system-defined conditions so that, should the condition be detected prior to the user having established a handler for it, there will be a guaranteed system-defined response. The user may, by a suitable library subroutine call, stack an entry to a default handler for a user-defined condition. Such stack entries, whether for system-defined or for user-defined conditions, are specially marked so they can be recognized. Once placed on the stack, they cannot be reverted, even though a call to <reversion> requests its removal. Later subsections elaborate on the subject of default handlers.

5.2.1.2 Signalling (Initial Implementation)

The chief purpose of saving a condition handler is to use it when and if proper notice is later given to do so. A call to <signal> is the act of serving this notice. For example, during execution of a procedure in ring s, we shall picture a call to <signal> of the form:

call signal ("condname", return_flag*, arglist_pointer);

The effect will be to invoke the handler whose entry datum is found in the topmost frame of the stack for condname, as found in <signals_s>. If this handler is the procedure named <proc>, then arglist_pointer is passed along to <proc> as an argument.

It is expected that the signal handling machinery just described will serve most needs of the typical user. The characteristic here is that handlers are stacked in and signalled from the same ring. Moreover, it may also be typical that the procedure identified as the intended handler will also be found in ring 32 (especially if it is a programmer-defined condition). In these situations locating and invoking the desired handler will be a relatively simple task. No protection problems will arise and hence there will be no intervention of the Gatekeeper. System overhead to perform the required services will therefore be kept to a minimum.

5.2.1.3 Establishing Default Handlers — User-Defined Conditions

A user is free to establish a default handler for any of his programmer-defined conditions by a call to a "sister" primitive of <condition> named <set_default>. (The basic MSPM reference is again BD.9.04.)

For example, executing the PL/I statement:

call set_default ("list_empty", refill);

while a process executes in ring k, will have the effect of stacking a default handler for the condition named "list_empty" in <signals_k>. This handler points to the procedure named refill. <Set_default> performs a service almost identical to that of <condition> in establishing the desired (default) handler. The only thing special about a call to <set_default> is that the stacked entry for this handler is specially marked to indicate that the entry corresponds to a default handler. <Reversion> will ignore a request to pop such an entry. Likewise, the Unwinder will not be able to revert this handler when and if called. The Unwinder's practice of reverting handlers is explained in Section 5.3.

5.2.1.4 Default Handlers — System-Defined Conditions

System-defined conditions are treated in a fashion which guarantees that a default handler will always be provided. (To simplify further explanation we shall

---

* If the procedure calling <signal> wishes to permit a return from <proc>, the value of return_flag must be set to 1 for the call and 0 otherwise. An attempt to execute a normal return from <proc> when return_flag is set to 0 will be interpreted by <signal> as a request to abort the process.

employ the generic condition name, "sys_cond".) Default handlers are provided on an as-needed basis. Thus, the default handler for "sys_cond" would be established in ring k only when the first reference is made to "sys_cond" in this ring during a call on <condition> or on <signal>.

For those interested in the details (the curious only), here is how it will be done.

An in-symbol definition for every system-defined condition will be preset in <signals_k . link> when (and if) this segment is created. Each such definition carries a trap pointer which we referred to in Chapter 2 as a trap-before-definition pointer. You can review this feature by a glance at Figure 5-2. Whenever called, <condition> (or <signals>) must find the offset within <signals_k> that holds the headword for the desired condition stack. To determine this offset <condition> (or <signal>) will call a useful library routine known as <generate_ptr>.* This routine will search <signals_k . link> for the preset definition that holds this offset i.e. value. Upon finding this definition, <generate_ptr> will note that the trap pointer is set (i.e., is non-null) and will then construct and execute a call to the special system-supplied trap routine whose name and arglist_pointer are designated via the trap pointer. The trap routine is designed to cause an entry to the desired default handler to be stacked in <signals_k> via a call to <set_default>. Upon return from the trap routine <generate_ptr> remembers to reset the trap-pointer (to zero), so any subsequent call to <condition> (or to <signal>) that references "sys_cond" cannot cause another default handler to be stacked in ring k. (Note, by setting different trap-before-definition pointers there is provision here such that system-defined conditions may invoke different default handlers in different rings. At present there is no specific application of this opportunity.)

5.2.2 Multi-ring Generalization (for future implementation)

The foregoing scheme for implementing condition handling gave us a brief picture of the basic plan for stacking and unstacking handlers, and for signalling conditions within a single ring. To picture the complicated cases which may be permitted to occur in a multi-ring environment we need a suitable model for ensuing discourse. We set the stage for such a model by presenting Figure 5-3. This figure attempts to show a snapshot of several stack segments and of <rtn_stk> for a hypothetical multi-ring process after a series of cross-ring calls has occurred (with no intervening returns). The procedures involved are assumed to reside in rings 33, 32, and 1.

*Defined in BY 13.02.

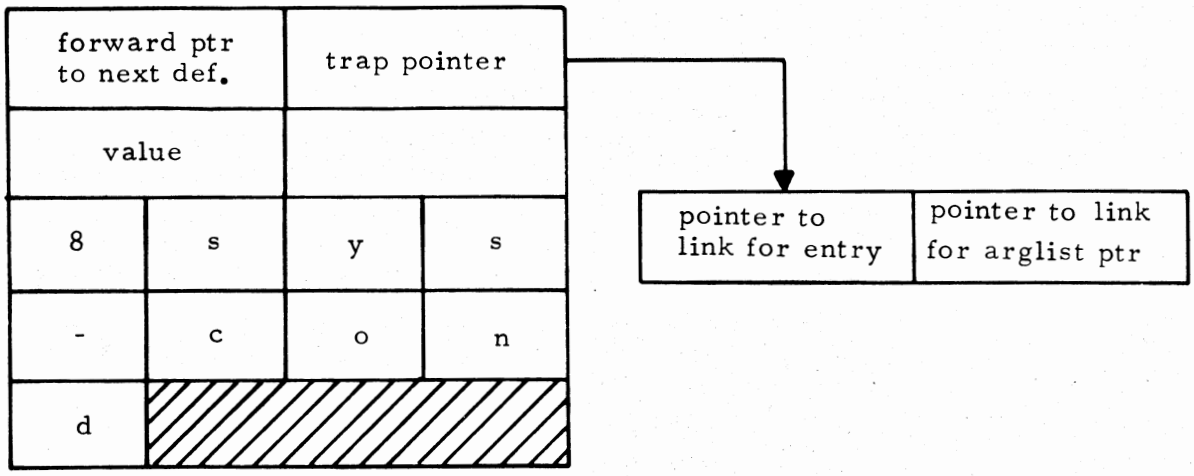
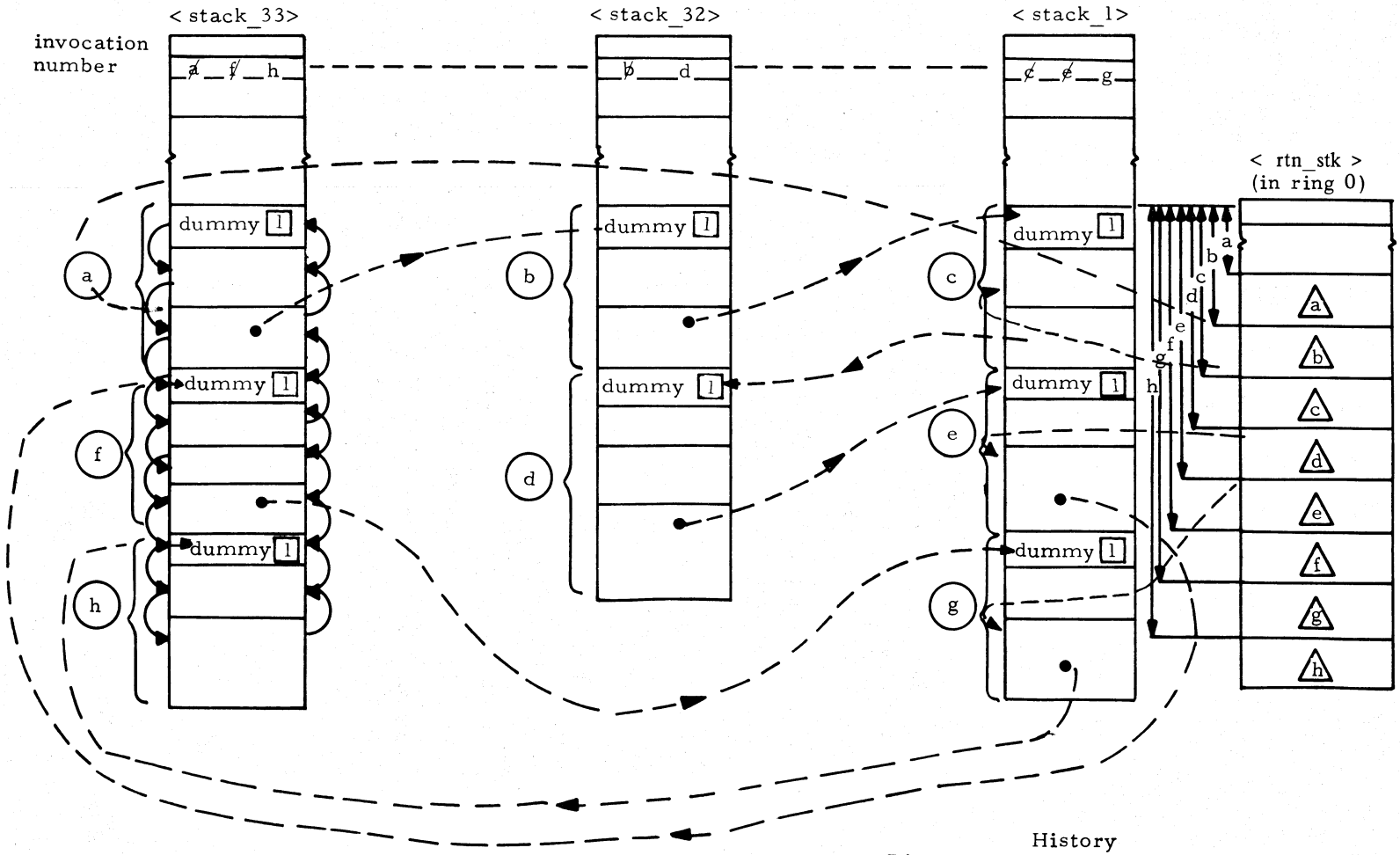


Figure 5-2. The Trap-before-definition Feature in an In-symbol Table Definition for System-defined Conditions

The trace can be seen to begin while executing procedures whose stack frames are in the group marked (a) in < stack_33>. Upon entering ring 33 for execution of these procedures, the invocation number* a was saved in < stack_33>|2 and a corresponding six-word entry of return information marked /a\ was threaded onto the return stack at < rtn_stk>|a. A "wall-crossing" call to a procedure in ring 32 then causes the invocation number b to be assigned to < stack_32>|2 and the entry marked /b\ added to < rtn_stk>. Note the cross-ring flag is shown marked [1] in the dummy (first) stack frame of the group marked (b) to be found in < stack_32>.

Control flows on from ring to ring, first to ring 1, then back to ring 32, etc. Each crossing into a ring numbered j updates the invocation number at < stack_j>|2. This is suggested in Figure 5-3 by showing older invocation values crossed out. The return information in each entry of < rtn_stk> is suggested schematically by the dashed (red) arrows from some of the entries (/b\ , /d\ , /f\ , and /h\ ) pointing to the respective stack frames of the procedures issuing the cross-ring calls.

*The invocation number concept was first described in Chapter 4 in connection with the description of < rtn_stk>, (Section 4.3.3).



Lower case letters for invocation numbers are intended to represent distinct ascending order integer values.

Notation: (k) signifies the < rtn_stk> entry whose invocation number is (i.e., stored at) k.

History

Ring crossing	Invocation number
33 → 32	b
32 → 1	c
1 → 32	d
32 → 1	e
1 → 33	f
33 → 1	g
1 → 33	h

Figure 5-3. A Trace through Seven Ring Crossings Employing Invocation Numbers b through h

We are now ready to look at details of the signal vector segments using specific examples based on Figure 5-3. While the invocation number was equal to b we suppose a ring-32 procedure executes:

call condition ("z", proc1);

Later, while again executing in ring 32, this time when the invocation number has the value d we imagine that another call on <condition> is executed:

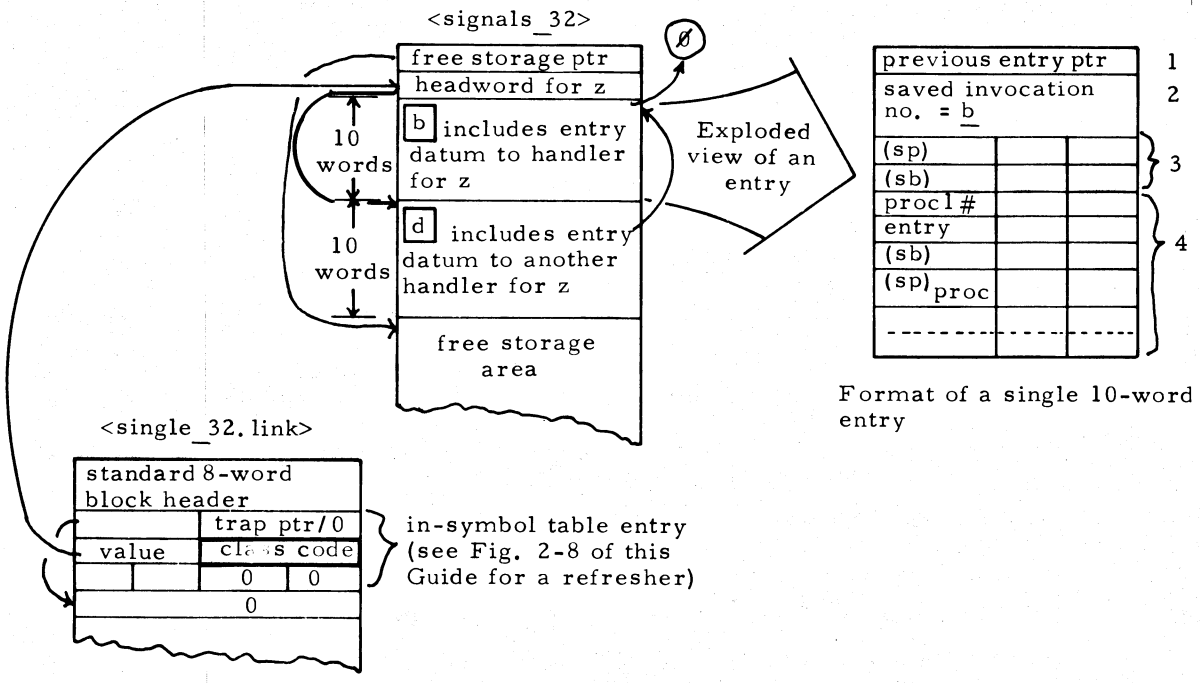
call condition ("z", proc32);

Figure 5-4 gives a somewhat detailed view of <signals_32> and of <signals_32.link> upon completion of this second call. Close inspection of this figure is warranted. Again we see how 10-word entry for condition "z" form a singly-threaded push-down list. The null back pointer in the bottom element is denoted by ⓟ. The detailed format of a 10-word entry may be surmised from examination of the exploded view for the bottom entry. The invocation number saved here is a copy of the current invocation number taken from <stack_32>|2 at the time <condition> places the entry in <signal_32>. We shall call this copy the saved invocation number, or sin. The stack pointer saved in words 3 and 4 of the entry correspond to the pointer for the procedure which called <condition>. This pointer is also used by the Unwinder procedure in its cleanup operation, an activity described in Section 5.3. The remaining six words constitute a standard entry datum for the procedure to be called when and if the <signal> procedure determines that this entry is for the currently active handler.

The figure also gives the format details for a typical in-symbol table entry or "definition": There is one such definition in <signals_n.link> for each push-down list of handlers in <signals_n>. Each definition gives the name of the condition and a pointer to the headword of the push-down list in the corresponding signals segment. One feature of the definition, the class code, is of special interest to the subsystem writer in connection with signalling. This feature will be discussed in due course.

Figure 5-5 elaborates by showing how handlers for several different conditions, e.g., "W", "T", and "X" would be stacked in <signals_1> and referred to from its linkage segment. A pointer to this free storage is maintained at word zero of the same segment.

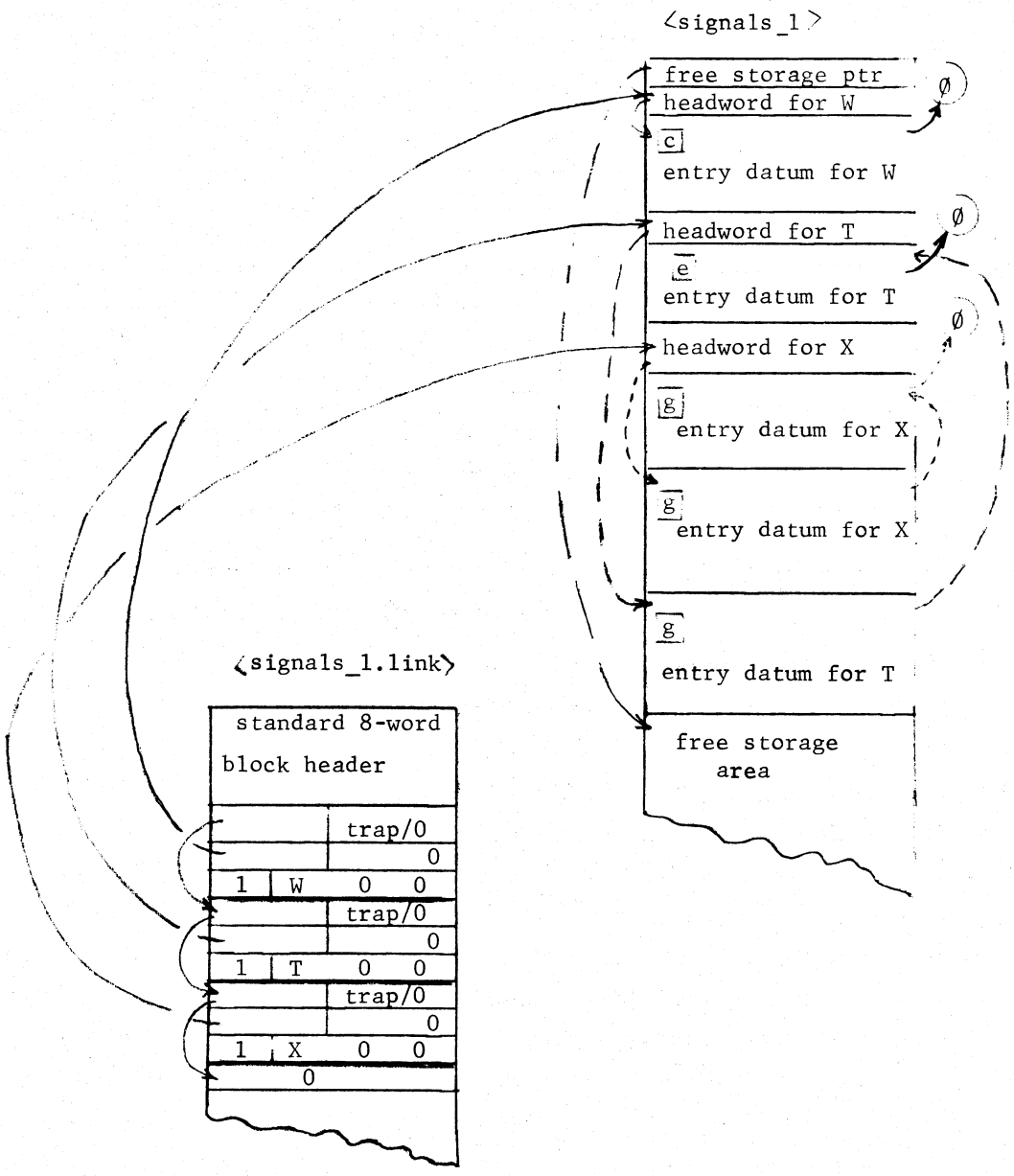
Figure 5-6 is a composite view showing snapshots of the <stack_n>, <signals_n>, and <signals_n.link> segments for the hypothetical process case history we began developing in Figure 5-3.



Explanation

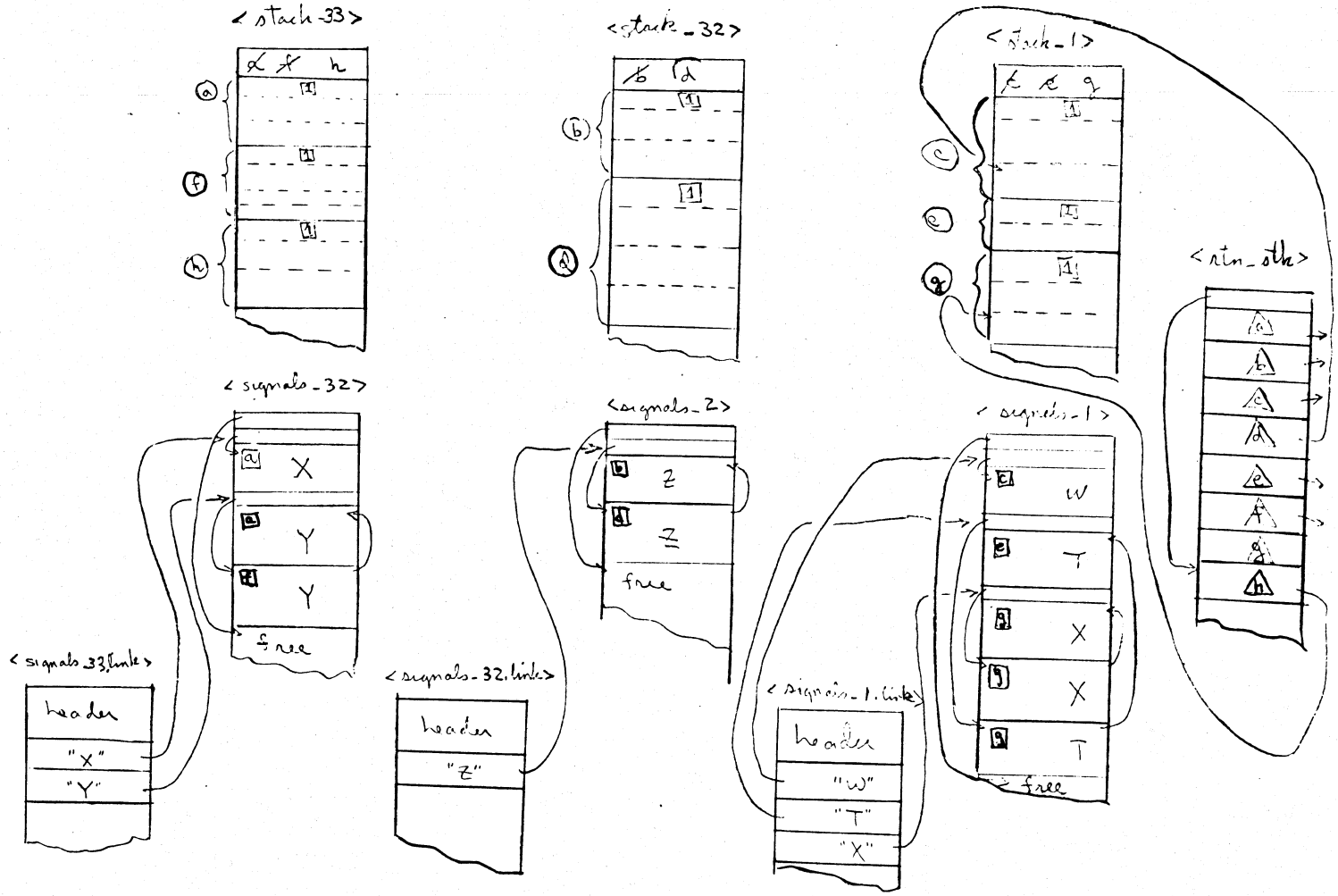
1. Previous entry ptr is null if in the first entry of a stack.
2. Saved invocation number (sin used by <signal>. (This number is zero for a default handler).
3. Pointer to stack frame for procedure which invoked <condition>, causing this entry to be created. (Used by the Unwinder). This pointer is zero for a default handler.
4. Entry datum for the handler. The stack pointer is used when the handler is an internal procedure. (See Chapter 3, pp 3-28.)

Figure 5-4. Details for a Signals Segment and its companion Linkage Segment



Also given are the associated linkage segment showing stacks for handlers of several different conditions. Each stack is pointed to by a different definition in the linkage segment.

Figure 5-5. Storage Structure for a Signals Vector Segment



Conditions are named "T", "X", "Y", and "W" in different rings.

Figure 5-6. Stacked Condition Handlers

The new display shows the stacked handlers for "X" and "Y" in ring 33 as well as those previously illustrated in more detail in Figures 5-4 and 5-5.

With this composite illustration we call attention to the fact that handlers for the same condition may be stacked in more than one signals vector as a result of calling <condition> from different rings. Notice, that handlers for "X" now appear in signals vectors for both rings 33 and 1.

5.2.2.1 Some Case Studies

We have now set the stage for considering various problems that <signal> must solve in locating the active handler. A set of illustrative case studies of graded complexity will be considered in this subsection. The discussion of each case shows how <signal> would accomplish its task. The bases for these cases and the principle or resolution employed by <signal> are summarized in Table 5-1. Probably, only a quick scan of this material is all that is justified on a first reading of this chapter.

In each of the cases below it is assumed that the signalling procedure calls <signal> while the former is executing in ring s. The current invocation number stored in <stack_s>|2 (and also at <rtn_stk>|0) shall be called curinv. All cases refer to Figure 5-6.

Case 1 - Simplest and most frequent — Active handler is topmost on its stack in the same ring as the signalling procedure.

s = 33
curinv = f

The call to signal is:

```
call signal ("Y", rtn_flg, arglist_ptr);
```

Upon being called, <signal> searches the definitions in <signals_s.link>, i.e., <signals_32.link> for a match on "Y", which it will indeed find in this case. The value associated with "Y" is used to locate the top entry for Y in <signals_33>. Upon comparing the value of the saved invocation number with curinv, a match ([f] = f) is found. The entry datum for the active handler is found in this matched entry, and <signal> then sets up a call to this handler. The details of the calling procedure need not be of concern to most readers, but are briefly described in Section 5.2.2.2 for the sake of completeness.

TABLE 5-1

Summary of Cases Illustrating <Signal>'s Search for an Active Handler

Case	s	Curinv	Signalled Condition	Active Handler Determined by <Signal>: Ring	Active Handler Determined by <Signal>: Saved Invocation Number (sin)	Remarks
1	33	f	"Y"	33	f	------
2	33	a	"Y"	33	a	One superfluous entry is popped to reach the entry for the active handler
3	33	f	"X"			
a				None found (search for "unclaimed-signal" handler).		Search of other signals segments denied
b				33	a	Assumes permissive class codes in <signals_i.link> permit a full search (see Table 5-3).
4	1	g	"Z"	32	d	Assuming permissive class codes in <signals_i.link>
5	33	f	"Z"	32	d	Assuming permissive class codes in <signals_i.link>

These five cases are based on the illustration in Figure 5-6, and are discussed in the text.

Case 2 - A variation on Case 1 — a superfluous entry must be popped off the stack in reaching the entry for the active handler — (an unusual case).

s = 33

curinv = a

The call to signal is again

```
call signal ("Y", rtn_flg, arglist_ptr);
```

Upon being called, < signal> would, as in Case 1, successfully find a definition for "Y" in < signals_33.link>, but this time the saved invocation number of the top entry in the Y stack of < signals_33>, [f], exceeds curinv. < Signal> perfunctorily pops this entry because the appearance of such an entry is regarded as a programmer goof (i.e., a failure to revert or pop an entry before returning from a procedure that stacked it). Using the backpointer in the popped entry for Y, < signal> then inspects the next entry in the stack for Y. Here a match is found between the sin, [a], and curinv, thus identifying the active handler. The setup for the call on the proper procedure follows normal procedures mentioned in Case 1.

Case 3 - Entry for an active handler is not found in the signals segment for the ring of the signalling procedure.

s = 33
curinv = f

The call to signal is:

call signal ("X", rtn_flg, arglist_ptr);

Upon being called, < signal> will locate the definition for "X" in < signals_33.link> which points to an entry (in < signals_33>) for which the sin,

[a] < curinv

In any one signals segment, the saved invocation numbers in a stack for a given condition must be in descending order from the top of the stack. The relation

[a] < curinv = true

then implies there can be no handler for X in < signals_33> for which

[f] = curinv.

The search of < signals_33> for a currently active handler, therefore, fails. Is there any use to look elsewhere? It's possible that a handler for X was established in some other ring for some prior invocation number. This might have occurred, for example, while executing in the ring immediately prior to crossing over to 33 while the invocation number was e, or in the "preceding" ring while the invocation number was d, etc. If so, a historical search of previous ring crossings and the corresponding signals segments might well turn up such a handler. However, freedom to "backtrack" through ring history may not always be the kind of search behavior we want < signal> to exercise. It is intended that the subsystem designer can, in fact, determine whether such search privilege is to be denied. This would be achieved by assigning

special values for the class code in the definition for "X" placed in < signals_33.link>. Class code values can be set in the definition when one uses the library subroutine

link_change (see BY.13.03)*

The various class codes and their significance in the control of < signal>'s ability to search for handlers will be discussed below. Depending upon the class code that < signal> finds in the in-symbol table definition for "X", two main possibilities arise:

Case 3a - Obtaining a default handler defined in the ring of the signaller.

The search of other signals segments is denied. A default handler will then be needed. Default handlers, incidentally, are recognized by virtue of their corresponding stack entries having a saved invocation number equal to zero. The stack for X in < signals_33> will therefore be searched backwards from its current top until an entry is found whose sin is zero.

As we have already seen in an earlier discussion, if X is a system-defined condition, there will always be a stacked default handler. If, on the other hand, X is a programmer-defined condition, it is entirely possible the programmer has failed to supply such a handler--perhaps through error. In such cases, < signal> after failing to find the default handler, then begins a search for the current handler of a system-defined condition known as "unclaimed_signal". A handler (at the very least a default handler) will always be found for this condition.

Case 3b - Searching for an active handler in signals segments of other rings.

The search of other signals segments is permitted. The retrace of ring crossings then proceeds in the following fashion: Examine the top (most recent) entry in < rtn_stk>. This entry holds the predecessor and also the prior invocation number (pin) in the form of a backpointer. (For a refresher, see Figure 4-16.) In our example, Figure 5-6, the predecessor ring would be 1, and the prior invocation number would be e. < Signals_1.link> would then be searched for a definition for "X". If found, the associated class code would be consulted before search of < signals_1> would be permitted to proceed. Then a scan of stack "X" in < signals_1> would be made in search of an entry whose saved invocation number is e, matching the value for what is now regarded as the prior invocation number. (In short, an active handler is found when sin matches pin.) If the search of < signals_1> fails, the search may be continued in a like manner after consulting the next entry in < rtn_stk>.

*Link_change has not been actually written and placed in the system library at the time this chapter was written.

For proper perspective, it's important to keep in mind that class codes are never consulted if the active handler is discovered during the first examination of the handler stack in < signals_s>. Class codes are consulted only under the following circumstances:

(1) After failure to find the active handler in < signals_s> when first examined,

(2) Prior to examining the stack of handlers in each signals segment (including a repeat attempt to look at < signals_s> during the backtracking operations.

Table 5-2 lists the various class codes and explains how each class code would be interpreted by < signal>. For example, if the class code for "X" were 13 or 14, search of < signals_1> would be denied and, in fact, the search would be terminated immediately. Moreover, if this were the case, the default handler would then be sought from < signals_s>, i.e., from < signals_33>.

If the class code were 12, then examination of < signals_1> would be denied, but the search would be permitted to continue to other signals segments by continuing the retrace of ring crossings in the manner just described.

Other class codes would also be interpreted as permission to search < signals_1>. In this example, such permission would not result in finding the active handlers in < signals_1>, but it would lead to the popping of the two handlers whose sins, [g] , exceed the current invocation value. Popping is justified because [g] exceeds e, the invocation number that was current while control was last in ring 1.

We can also make one other observation from a study of Table 5-2: Case 3a will occur only when the class code found in < signals_s.link> is 11, 12, or 13. Case 3b can occur when the class code in < signals_s.link> is any other value.

We should now have enough familiarity with the signal searching rules to see how the Case 3b search would terminate under a variety of possibilities for class code values in the various < signals_i.link> segments. Thus, if all class codes were = 0, the active handler for X would eventually be found to be the one marked [a] in < signals_33>, as a result of retracing back through five < rtn_stk> entries until the one marked △b which identifies ring 33 is found as the predecessor and a as the prior invocation number.

This ring-by-ring search is summarized in Table 5-3.

TABLE 5-2

Class Codes and their Interpretation during Search for an Active Handler

Class Code Value	Interpretation
0	No search constraints. O.K. to search through the stack in this signals segment and to proceed, if necessary, to the next signals segment as determined by a retrace of ring crossings.
11	May search the stack in this signals segment. If active handler is not found here, terminate the search by taking the default handler indicated in this segment. If no default handler (programmer-defined condition only) is found, return to < signals_s> and begin a search for the currently active handler of "unclaimed_signal".
12	Do not search in this signals segment, but proceed to search in signals segments of other rings by retrace of ring crossings. (Note: If this is the signaller's signals segment (< signal_s>), we've already looked here once before beginning the retracing process.)
13	Do not search in this signals segment. Terminate the search, immediately. Employ the default handler indicated in the signals segment of signaller's ring s. If no default handler is found there (programmer-defined condition only), begin a search for the active handler of "unclaimed_signal." Note if this code were found in < signals_s.link>, the search would be limited to < signals_s> exclusively. Default handler would, if needed, always be taken from < signals_s>.
14	Identical to 13 with one exception. If 14 has not been found in < signal_s.link>, and if the active handler is not located after searching in < signals_s>, permission to begin the ring-to-ring retrace is granted. However, search will terminate immediately with same effect as a code = 13 if this code 14 is ever again encountered.
Others	For signal searching purposes other class code values are interpreted as if they were code 0, i.e., no constraints.

TABLE 5-3

Details of Ring-by-Ring Search

Ring i whose <signals_i> is being considered	Associated invocation number	Assumed class code for "X" in <signals_i.link>	Effect (see key below)	Designation of referenced <rtn_stk> entry	Saved values in designated <rtn_stk> entry	
					predecessor ring number	prior invocation number (pin)
33 (= s)	f (curinv)	0	1	f	1	e
1	e	0	1, 2	e	32	d
32	d	0	3	d	1	c
1	c	0	1	c	32	b
32	b	0	3	b	33	a
33	a	0	4			

KEY:
1 Entry for an active handler is not found.
2 Entries marked [g] are popped because g > curinv = e.
3 "X" not found in <signals_i.link>. Proceed with retracing.
4 Entry for active handler is found, i.e., curinv = [a] .

Other possible class code values for "X" in < signals_33.link> and/or < signals_1.link> could determine a different active handler for X. As an exercise, you could verify the results given below for the stated combination of class code values assumed to be associated with "X".

Class code values in < signals_33.link>	< signals_1.link>	Active handler
0	12	Active handler is the one marked [a] in < signals_33>. (Note however < signals_1> will not be examined, so the handlers marked [g] will not be popped.)
12	12	Active handler will not be found upon first search of < signals_33>. Thereafter, < signals_33> (and also < signals_1>) will be skipped over during retrace through < rtn_stk>. When all entries in < rtn_stk> have been used up in the retrace procedure, the search will be declared a failure and a search for X's default handler in < signals_33> will be made. If no default handler for X is present in < signals_33> (this is the case illustrated in Figure 5-6), a search for the "unclaimed_signal" active handler will then be made.
0	11	Entries marked [g] in < signals_1> are popped. Failure to find active handler upon first inspection of < signals_1> terminates the search. A default handler is then looked for in < signals_1>. If not found, search is begun for "unclaimed_signal" active handler.
13	0	Active handler will not be found on first examination of < signals_33>. Retrace permission will then be refused, forcing a search of < signal_33> for a default handler (and eventually for "unclaimed_signal" handler).
14	13	Same net effect as preceding example. Active handler will not be found on first examination of < signals_33>. Permission to retrace will then be granted but upon retracing to examine < signals_1> a code 13 will be encountered, terminating the search and forcing the use of default handler in < signals_33> etc.

Case 4

s = 1

curinv = g

The call to signal is:

call signal ("Z", rtn_flg, arglist_ptr);

Upon being called, < signal> searches < signal_1.link> without success. No definition for "Z" can be found there. Assuming the class code for "Z" in the various < signals_1.link>s permit inspection of the respective < signals_i> segments, then < signal> will discover the currently active handler to be the one whose entry is marked [d] in < signals_32>. This discovery will be made after retracing through the < rtn_stk> entries marked [f], and [e], and getting a match between sin = [d] and pin (= d). (Of course, if the class code for "Z" in < signals_32.link> is 12, 13, or 14, the search would be terminated for one reason or another without ever identifying the entry marked [d] as representing the active handler. A default handler for X would be sought in < signals_1>.)

Case 5

s = 33

curinv = f

The call to signal is:

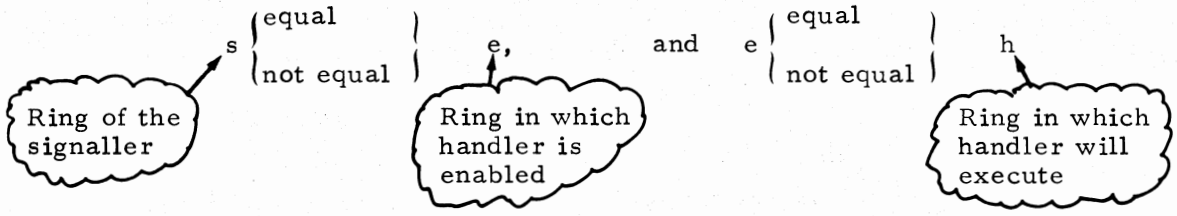
call signal ("Z", rtn_flg, arglist_ptr);

This case is very similar to that of Case 4. The entry marked [d] in < signals_32> may be discovered to represent the currently active handler, provided search of < signals_32> is permitted by the class code for "Z" found in < signals_32.link>. However, if search permission is not granted, a default handler must be sought from the ring of the signaller which, in this case, is ring 33. Further comparison of Cases 4 and 5 is given in the next discussion.

5.2.2.2 Invoking the Handler Procedure

We have now seen two cases where an entry for a handler stacked in one ring (ring 32) is discovered as the active handler for a condition signalled from another ring (ring 1 in Case 4 or ring 33 in Case 5). Recall that in Cases 1, 2, and 3 we considered mainly situations where the active (or default) handlers were found in the same ring as the signallers. In either situation, there is a further possibility that

the handler itself resides in still another ring. In short, a signalling procedure, < signaller>, in ring s may have the effect of involving a handler, < p> in ring h via a condition stack entry in the enabling ring e. Because

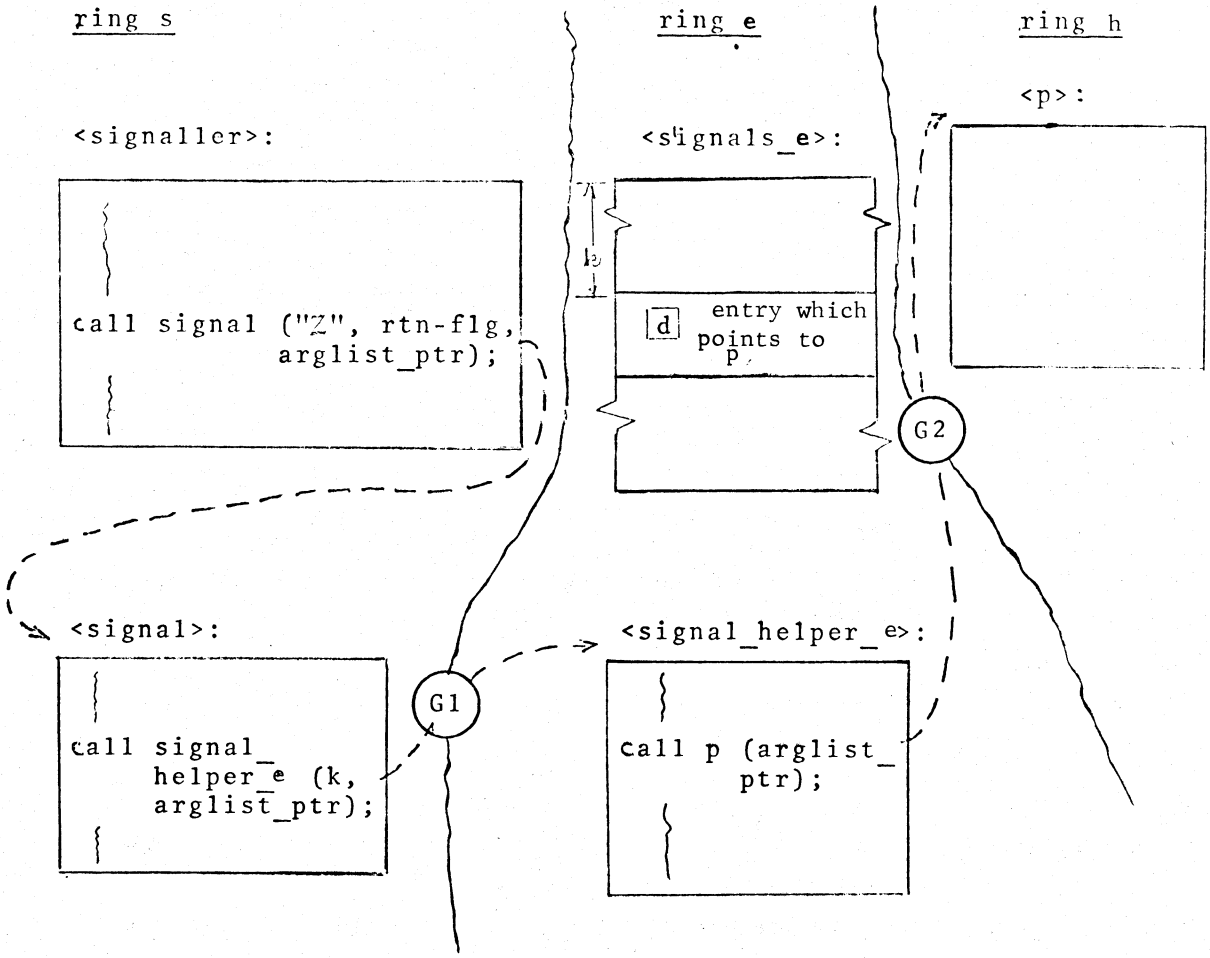


none, one, or two ring changes may be involved in signalling a given condition and invoking the desired handler.

Of course, every such call on < p> should be covered by existing Gatekeeper protection mechanisms. Thus, if < signaller> were calling < p> directly, with arglist_ptr as an argument, we would expect to obtain Gatekeeper intervention as needed. We expect no less protection in the case where < p> is invoked "indirectly" by < signaller>. In addition, we want also to assure that < p> is callable from the ring, e, in which it was established as a handler. The principle here is that a procedure < E> should not be allowed to force more privileged procedures to use a handler which < E> itself is not privileged to invoke directly.

These controls are partly achieved by forcing < p> to be called from ring e with the aid of a special "helper" segment in ring e named < signals_helper_e>, as suggested in Figure 5-7. When < p> is called from the enabling ring e in this way, and when this ring differs from the ring of < p>, i.e., $e \neq h$, the ordinary Gatekeeper intervention can be counted on to guarantee that no invalid ring crossing is being attempted here. We will also see how the Gatekeeper can be counted on to validate the arglist_ptr being passed to < p> from < signaller>.

< P> should always be called from the enabling ring e, even when $s \neq e$. Here is how this subtle but necessary bit of control is assured. Once having located the active handler, < signal> issues a call to < signals_helper_e> (in ring e), passing to it as arguments the entry datum for the active handler and the arglist_pointer parameter. This call on < signals_helper_e> forces a desired intervention by the Gatekeeper when $s \neq e$. The intervention is desired to validate the arglist_ptr. The intervention is guaranteed because: (a) The ring bracket for < signals> is (1, 63, 63), which means < signal> always executes in the ring of its caller, and (b) the ring



Key:

G1 means the Gatekeeper intervenes and validates arglist_ptr during the permissible ring crossing from s to e ($s \neq e$). <Signal >'s ring bracket is (1, 63, 63) while <signal_helper_e >'s ring bracket is (e, e, 63).

G2 means the Gatekeeper intervenes again, this time to protect against a possibly illegal ring crossing from ring e to ring h.

Figure 5-7. Mechanism used to Invoke an Active Handler

bracket for < signals_helper_e> is (e, e, 63), which means < signal> may call its helper from any ring. (The next subsection will show how < signal> gets the help it needs to search in and/or modify signals segments of other rings.)

5.2.2.3 Ring Brackets for < condition>, < reversion>, and < signal>

These three system primitives will in general be called from procedures in any user ring and from any administrative ring (except ring 0). Moreover, no ring crossing overhead will be incurred when one of these primitives is called, because the ring bracket for < condition>, < reversion> and < signal> are each (1, 63, 63). Since the access bracket is (1, 63), each of these primitives executes, when called, in the ring of its caller.*

Ring privileges required during signal search

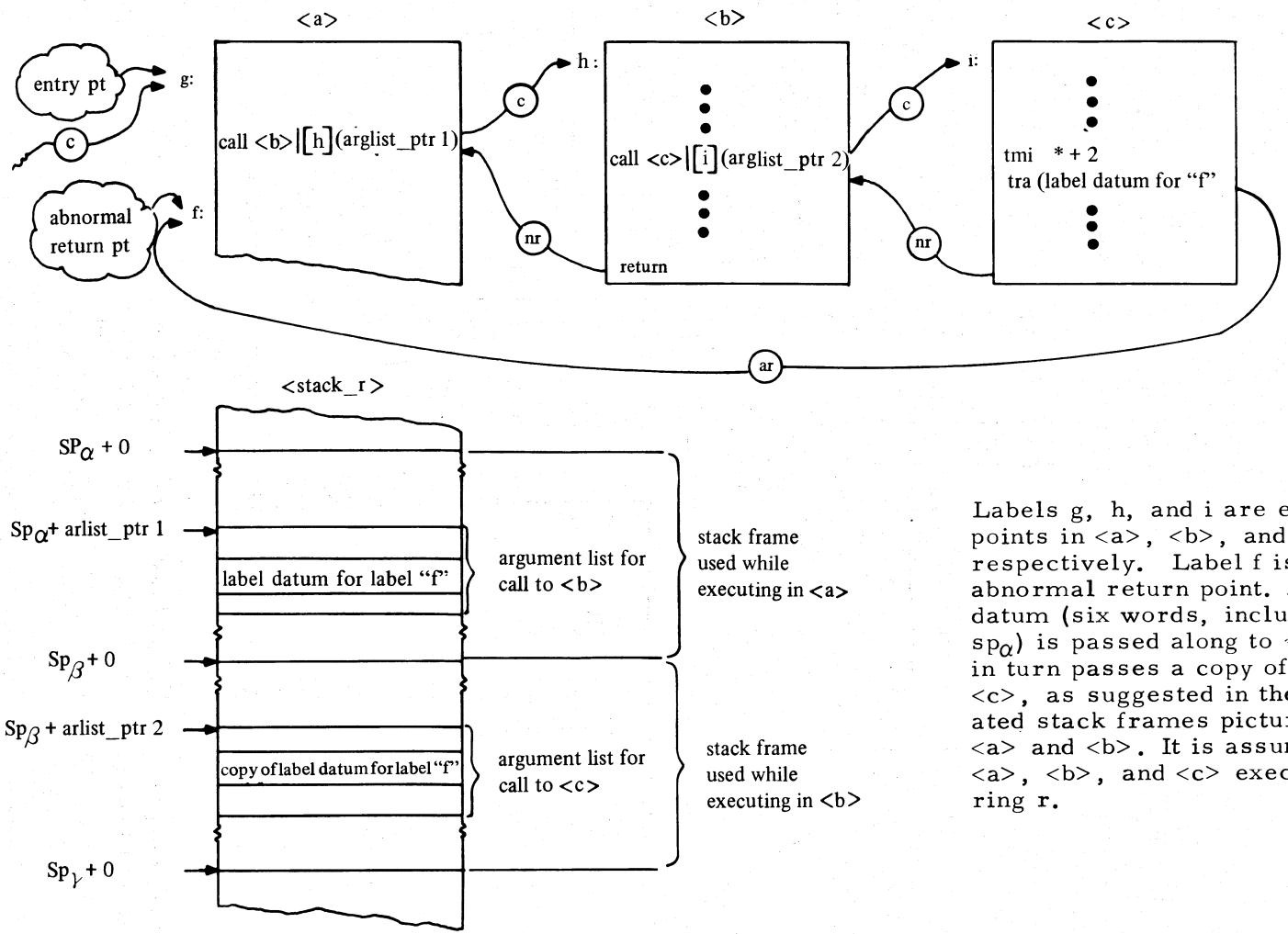
When, during its search, < signal> fails to find the active handler in < signals_s> then, class codes permitting, signals segments in other rings must be searched. Search of a < signals_i> segment implies both read and write privileges. Suppose some of these signals segments may be in rings r such that s > r. How can < signal>, executing in ring s, make read or write data references to segments in inner rings? In the Multics solution detailed in BD 9.04, you would see that < signal> in fact calls on a special auxiliary routine named < signal_search>. It is this routine that continues the search when other signals segments must be inspected. < Signal_search> is a special ring 1 procedure whose call bracket is (2, 63). Naturally, if $s \neq 1$, then a ring crossing will occur when < signal> calls to or gets a return from < signal_search>, causing Gatekeeper intervention. Thus, at most one pair of ring crossings (call and return) will be involved in the typical use of <signal> for finding an active handler.

5.3 ABNORMAL RETURNS — ADDITIONAL DISCUSSION

5.3.1 General Concepts

This section is intended to provide background concepts that will lead to a fuller appreciation of the Unwinder. A procedure may have one or more entry points and none, one or more abnormal return points. Here we review the distinction between an entry point and an abnormal return point. Figure 5-8 will help develop both the differences and similarities.

*We will see in Chapter 6 that for such a procedure a separate copy of its linkage segment will be maintained in the process for each ring in which the procedure is called.



Labels g, h, and i are entry points in <a>, <b>, and <c>, respectively. Label f is an abnormal return point. A label datum (six words, including sp_α) is passed along to <b> which in turn passes a copy of it to <c>, as suggested in the associated stack frames pictured for <a> and <b>. It is assumed that <a>, <b>, and <c> execute in ring r.

Figure 5-8. Distinguishing Between Calls (c), Normal Returns (nr), and Abnormal Returns (ar)

Even though, from a user's point of view, entry and abnormal return points appear syntactically similar, and in fact may even have identical storage representations, there is a distinct functional difference. The difference has to do with the availability of needed information when control passes to one of these points of a procedure. When entered via a call, a procedure should at that instant require no information other than what it is capable of developing and what is passed to it in the form of an argument list. Under these circumstances a procedure can and always does begin functioning with a new stack frame. On the other hand, when control passes to a procedure via an abnormal return point, execution resumes. This implies that certain information necessary to this resumption of effort may have been previously accumulated, probably in its then current stack frame. Therefore, resumption at an abnormal return point in the general case clearly forces the need to recover this stack frame, i.e., reset the stack pointer to this frame. The clerical details involved in resetting stack conditions, and in recovery of space for all allocated temporary data in the intervening procedures, are numerous. Even if all procedures in the chain of calls being bypassed (including the two procedures at the end points of the chain) have executed in only one ring, the complexity is sufficient to justify a system-provided Unwinder service. If we consider the more general case where procedures in the call chain may have executed in different rings, the clerical complexity is not only compounded, but protection issues dictate that a ring-0 Unwinder is required.* In the current implementation of the Unwinder, the more limited "single-ring service" is all that is provided. Our discussions in this section are based on the current design for the more general multi-ring service. The general service is justified on the grounds that the Multics user should not be forced to be conscious of ring crossings in planning an abnormal return. In many instances, a user may not even know about ring crossings in the path of the abnormal return. The next paragraphs will indicate more specifically some of the complexities that are involved.

Let s be the ring of < a> and let us assume that the pointer to the desired frame in < stack_s> is part of the label datum used for generating the abnormal return in a statement of the form:

```
tra  label_datum
```

(We can normally assume that the value of label_datum has been passed along the call chain as an argument.) One might then imagine the abnormal return to < a> can

*The principal MSPM reference is BD.9.05.

be achieved by executing some kind of return sequence that includes the appropriate adjustment of the stack pointer at <stack_s>|2 and the base address registers. Adjustment of <stack_s>|2 would have the virtue of recovering space in <stack_s> for frames of ring-s procedures that are bypassed in this return.

The label datum that defines the abnormal return is not necessarily "authentic", however. Suppose, for instance, the stack pointer in the label datum has been inadvertantly altered by the user and no longer corresponds to the beginning of any stack frame in <stack_s> (let alone to the frame that was intended). Clearly, chaos would result if an attempted abnormal return were allowed to proceed using an incorrect stack pointer. To check the given stack pointer for validity will involve, at the very least: 1) a search through the back pointers in the stack frames (at sp|18) for one that matches the given stack pointer, and 2) provisions for error returns in case the search fails to turn up a "good" match.

Even if all goes well, however, two very undesirable side effects must be considered. These would occur if any rings were crossed in the chain of calls from <a> to the point where the abnormal return was invoked. Specifically, suppose the chain is: <a> calls <b> calls <c>, and suppose each call involves a ring crossing. At <c> we imagine the abnormal return is invoked by executing a statement like

tra (label_datum for "f")

as suggested in Figure 5-8.

Side effect No. 1. Suppose we fail to pop the top two frames in <rtn_stk> while executing this return. What will be the consequence the next time a normal return is executed that involves leaving ring s to reach an antecedent of <a> ? For example, suppose <a> was originally reached at <a>|[g], via a call from ring r. In attempting to oversee the normal return from <a>, the Gatekeeper expects to find a validating return address in the top frame of <rtn_stk>. This address will not be found, because the frame in question is now buried below the top of <rtn_stk>. This failure causes the Gatekeeper to signal an unrecoverable error. The difficulty could be avoided only in very special situations where one could guarantee that all of <a>'s antecedents are in ring s.

Side effect No. 2. What about the other stack segments that hold frames for bypassed procedures? If we fail to pop these frames while executing the abnormal return, then space involved becomes unreclaimable.

Figure 5-9 suggests why the frame for <b>, pictured in <stack_t>, and any other frames that may have been stacked during the last "visit" to ring t (cross hatched), can never be reclaimed. The pointer at <stack_t> |[2] will not have been altered. Thus, after the abnormal return to <a>, any future visit to ring t will force the adding on of additional stack frames beginning at the place marked "next". As a matter of fact, since we have also failed to pop the appropriate frames in <rtn_stk>, all space thus far used in <stack_t> would be unreclaimable.

5.3.2 The Unwinder Details

The Unwinder mechanism is provided to perform two principal classes of service:

(1) To validate or screen an attempted abnormal return and, if valid,

(2) To perform various clerical tasks, including those motivated in the preceding subsection, which might otherwise be left undone when normal returns are bypassed.

In subsequent subsections we elaborate on these points.

5.3.2.1 Validation of the Abnormal Return

For this discussion we again employ as an example the case where <a> calls <b> calls <c>, with <c> attempting to execute an abnormal return to <a> |[f]. There are two crucial reasons for validating this abnormal return.

(1) The ring access rules which were developed for entry points also apply to abnormal return points. Thus if s, t, and u are the ring numbers of <a>, <b>, and <c> respectively, and if u lies outside the call bracket of <a> an abnormal return from <c> to <a> should be illegal. The Unwinder should be (and is) held responsible for making these ring access determinations.*

(2) Having determined that <c> has ring access to <a>, and in the event <c>'s ring lies within the call bracket of <a>, it is also necessary to verify that [f] is, in fact, an anticipated reentry point.

If <a> has been coded in a higher level language like EPL or EPLBSA, then every such location will be so declared. These declared reentry points are referred to as doors. In declaring that [f] is a permitted abnormal return point, i.e., a door, it is assumed that the author of <a> is anticipating inward returns

*The Unwinder, in fact, calls a ring-0 procedure named <get_ring> (see BG.3.01) for the help in accomplishing this check.

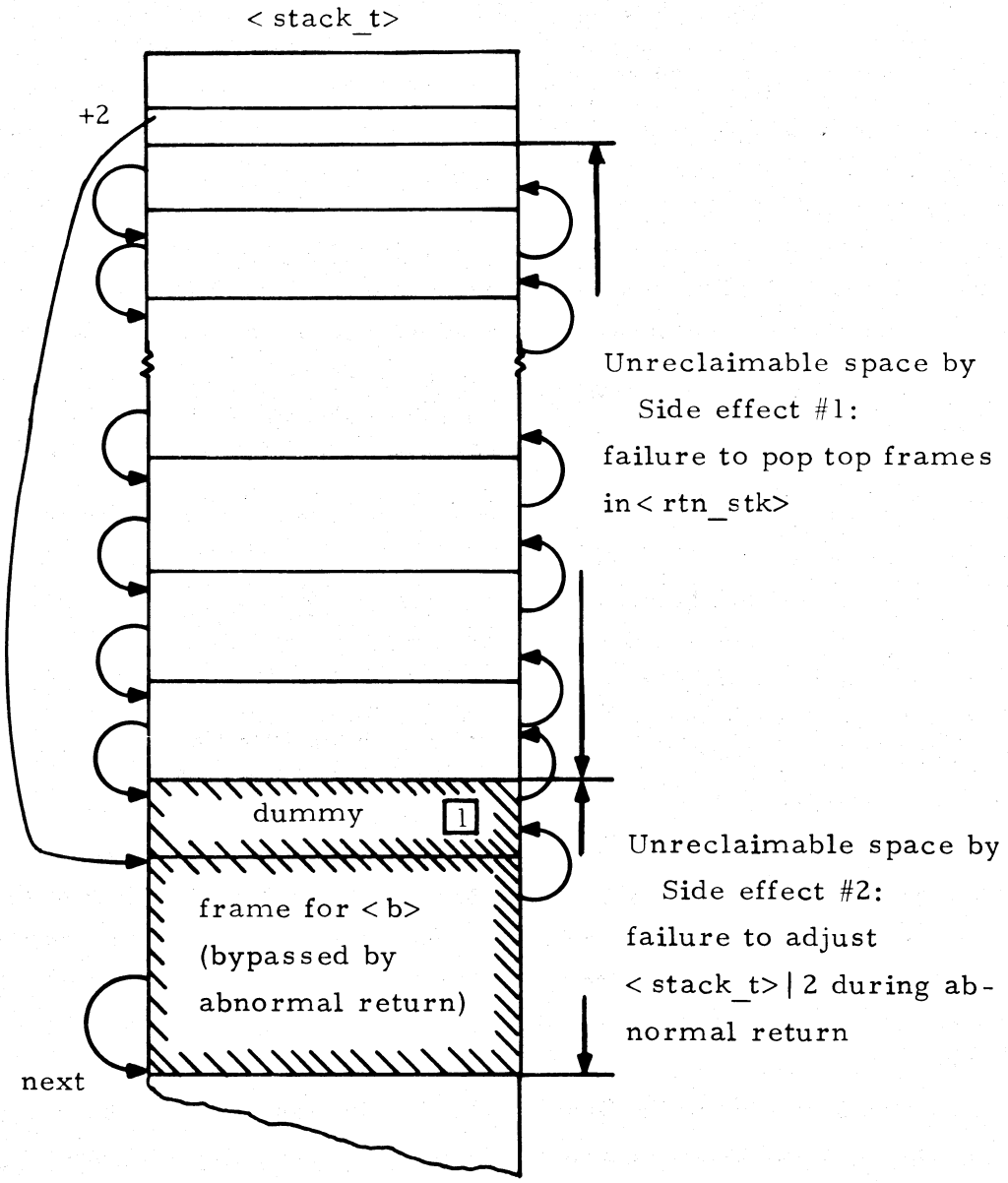


Figure 5-9. A Picture of < stack_t >

at [f] and has presumably programmed accordingly. If [f] is not named a door, the Unwinder should presume that an inward return to this point is not anticipated and cannot be risked, i.e., is likely to result in undesirable, even chaotic behavior.

Such a design philosophy is essential for the protection of supervisor routines that expect abnormal returns. Extending this concept to user-constructed subsystems implies that every abnormal return point designed to take control from an outer-ring procedure must be marked or declared by its author in the source code.

From such declarations the translators can generate doors in the linkage section in the form of specially formatted entry points. The format for a door is similar to that of a gate. Refer to Figure 4-23.

<Unwinder> will treat the word pointed to in the no-op instruction of the entry as door information, rather than as gate information. Only the first bits of this word are of interest to the <unwinder>, bits 6 and 7, called the "g" field. A value of g = 2 identifies this entry as a door.

Some of the processors within the Multics system e.g., EPL) will provide* for establishing doors at the users' request.

[Here will follow in some future revision of this document an example of such a declaration to be used in EPL.]

Some of the same processors will also interpret all non-local go to statements as abnormal returns. The generated code in each such instance is a call to the Unwinder. Thus the EPL statement:

```
go to a$f;
```

will result in generated code equivalent to

```
call unwinder (a$f);
```

A subsystem writer who codes in a programming language that does not have this feature must, of course, "manually" call <unwinder> when executing an ababnormal return.

*Not yet implemented in EPL as of 1/1/69.

For this reason it is worthwhile to explain briefly the protection provided in Multics in case a programmer fails to employ < unwinder > when attempting to execute an abnormal return. The design concept here is this: Only if the abnormal return is to an inner ring procedure is system intervention mandatory. This intervention will prevent a user from damaging either more sensitive procedures within a subsystem or the supervisor itself. Each user should be given the freedom to do what he wants to (or thinks he can do correctly at his own risk, to avoid the unnecessary system overhead) with procedures in rings he has full control over. With this philosophy in mind we see that an abnormal return, executed without the aid of < unwinder >, can be regarded more or less as a disguised call. If an inward crossing is attempted, the Gatekeeper should and would in fact intervene. Of course, the Gatekeeper would then properly interpret this transfer as an inward call. We are reminded that every inward call must be verified by determining that it is a gate (g = 1 in gate_info). If a valid abnormal return point is properly declared as a door, (g = 2), the Gatekeeper which is in search of a gate, will necessarily recognize the discrepancy and sound the alarm. It should now be clear why in the Multics design gates and doors are necessarily mutually exclusive.

5.3.2.2 Handling Unfinished Business

< Unwinder > takes full responsibility for reverting the stack frames of bypassed procedures. It also reverts frames in < rtn_stk > when and if ring crossing(s) have been involved. The reversion is achieved by tracing backward through the chain of stack frames that correspond to the pending returns. The backward search ends when a stack frame is reached whose address matches that given in the return label which has been passed to <unwinder> as the argument. < Unwinder >, because it is in ring 0, is able to consult the top frame of < rtn_stk > when a dummy frame, indicating a ring crossing, is encountered. Each stack frame or < rtn_stk > frame is reverted as it is passsed over in this scan for the matching stack address.

Cleanup Concepts

Are there other types of temporary data storage besides stack frames which also should be reverted when normal returns are bypassed? Indeed there are--in some subsystems, as we shall see later in this subsection. Multics must be prepared to serve such subsystems. < Unwinder > is endowed with a built-in capacity

of supervising the recovery of such other temporary storage when and if the subsystems programmer wishes this service to be performed. This type of activity is referred to in MSPM as "cleanup".

In subsystems that are coded in EPL, for instance, two kinds of temporary data (i.e., recoverable storage) are subject to cleanup. These are:

(1) Reversion of unwanted condition handlers, and

(2) Recovery of automatic storage for data types that must be kept on free storage lists outside the customary stack frame (e.g., arrays of varying strings whose datum portions are kept in a special free storage segment called <free_>.)

Other types of recoverable data may arise and be recognized in the particular subsystem you design. What follows in the next paragraphs is a brief outline of the general cleanup mechanism that has been embedded within <unwinder>.

Cleanup activity is regarded as a special task to be invoked, when needed, in connection with any (or with each) pending return that's being bypassed in the course of the unwinding process.

Each cleanup task is invoked as if it were a signalled condition.

The Unwinder has, in other words, been designed to behave as if it has been signalled to perform the requisite cleanup task, if any, on behalf of each bypassed procedure. To implement this signalling analogy, a handler for each cleanup activity is stacked in the format of a bonafide condition handler under the condition name "cleanup". This word, incidentally, is specially reserved by Multics for this particular use. The user is allowed to stack and revert handlers for "cleanup", via calls to <condition> and <reversion>, but he may not signal "cleanup", i.e., <signal> will reject a call of the form:

```
call signal ("cleanup", etc. 1,  etc. 2 );
```

As each stack frame is about to be reverted, <unwinder> first consults the corresponding <signals_i> segment for a "cleanup" handler having a matching invocation number and stack pointer. (In this search <unwinder> performs a task quite similar to that of <signal>.) If no such handler is found the stack frame now being considered is reverted and the unwinding process continues. If an appropriate "cleanup" handler is found, unwinder generates and executes a call to the designated cleanup procedure before reverting the stack

frame. The job that a properly written cleanup procedure must then accomplish is:

(1) Via calls to <reversion>, revert all as yet unreverted handlers that were stacked while executing the procedure being bypassed. Also, revert this particular cleanup handler.

(2) Free all space occupied by "automatic" data that was previously allocated to free storage lists.

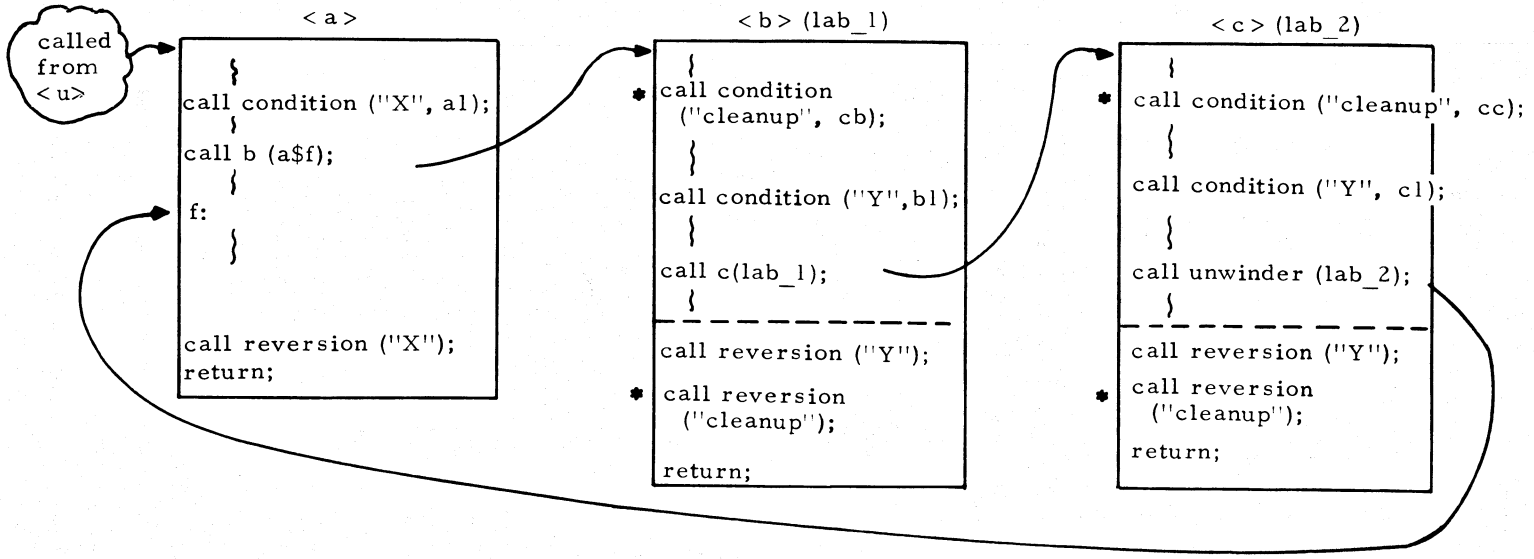
Who writes the cleanup procedures ?

From the above discussion we can see that writing and using cleanup procedures can become a pretty tricky business. Their main function of course is to prevent undue growth of "dead" storage in a process. A user may write his own cleanup routines and establish them as condition handlers – as many as he wishes. On the other hand, writing cleanup procedures and seeing to it they become condition handlers (by calls to <condition>), and then later reverting them in the event the abnormal return never gets executed, is the sort of mechanical programming we would normally want compilers or assemblers to generate for us wherever possible. EPL, fortunately, is one of those compilers that offers some of these cleanup services. However, even with EPL, the user, when programming abnormal returns, is expected to program his own cleanup routines for freeing up space that has been previously reserved by "allocate" statements. We shall elaborate momentarily on the services offered by the EPL compiler.

First, we summarize the various reversion steps of < unwinder >, and the reversion accomplished as a result of cleanup handlers. These are pictured schematically in Figure 5-11 for the abnormal return situation depicted in Figure 5-10.

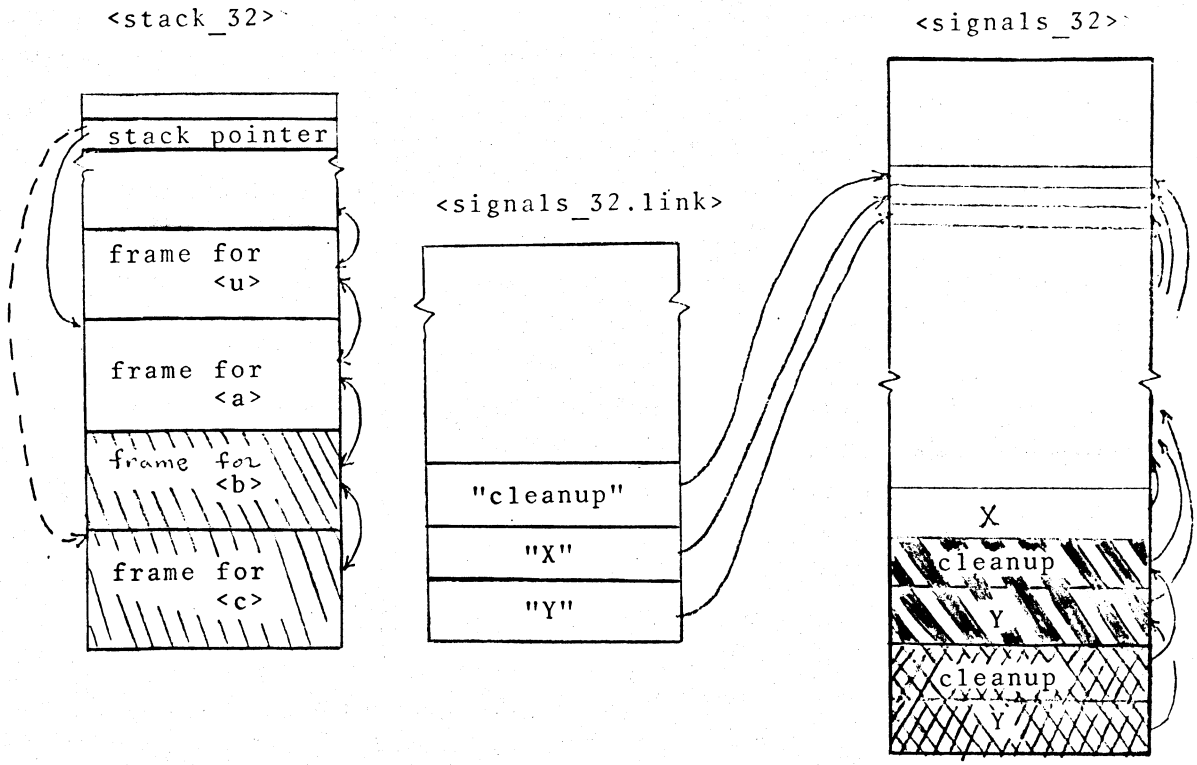
We again consider the abnormal return from <c> to <a>, bypassing <b>. The EPL-like coding displayed in Figure 5-10 shows the stacking of cleanup handlers during execution in <b> and in <c>. The case also presupposes other calls to <condition > during execution of <a >, <b > and <c> for the conditions named "X" and "Y", as shown. For simplicity we imagine first that all these procedures reside in the same ring (32).

Figure 5-12 is provided to suggest, by contrast with Figure 5-10, the nature of the cleanup service offered by the EPL compiler. Figure 5-12 gives an itemized list of the "free services" generated by EPL. It will be noted that no explicit reference to cleanup procedures or even to <unwinder > is required when writing code

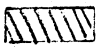


The abnormal return bypasses calls to < reversion> (shown below dashed lines in < b> and < c>, which would be executed if normal returns were taken. The coding in this figure shows explicit EPL – like calls to < condition>, < reversion> and < unwinder> using a compiler that does not offer special cleanup service. The programmer would also be required to supply the cleanup procedures referred to as cb in < b> and cc in < c>. Figure 5-12 shows the same case coded in EPL, where starred calls in < b> and in < c> are no longer needed.

Figure 5-10. Chain of Calls < a>→< b>→ < c> and Abnormal Return to < a> | [f]

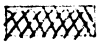


Key:



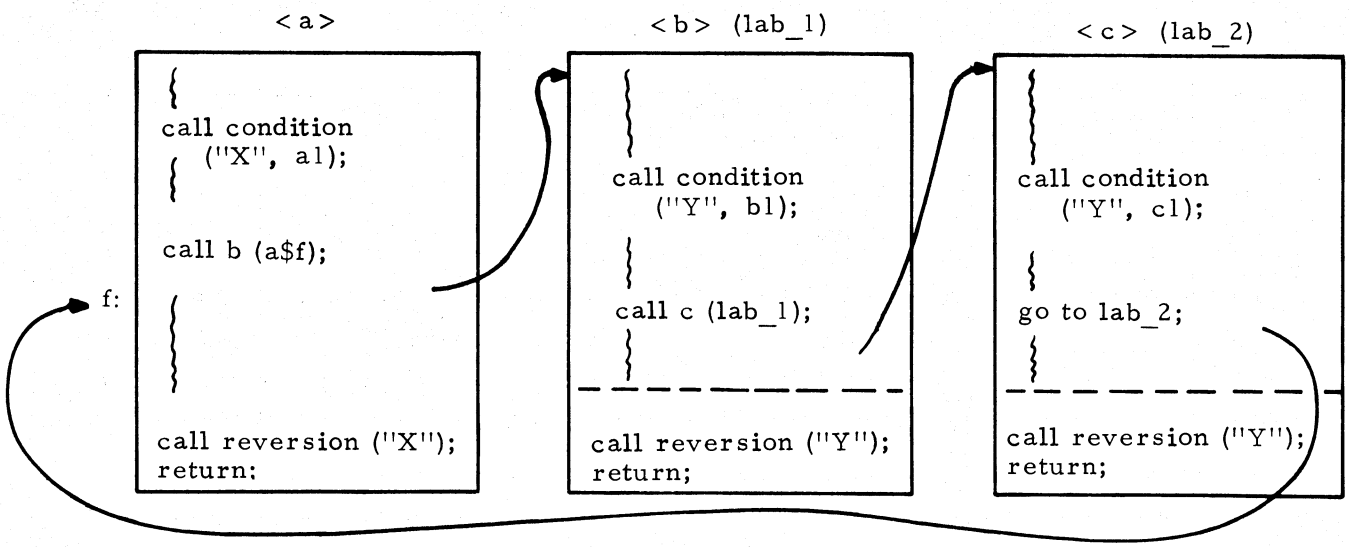
reverted by the <unwinder>

reverted by cleanup procedure cb



reverted by cleanup procedure cc

Figure 5-11. Stack Frames and Handlers Reverted After Successful Return from Call to < Unwinder >



Note:

(a) No specific call to establish handlers for "cleanup" are needed. These are generated by EPL and placed in the so-called prologue of the target code. The prologue is called as an internal procedure.

(b) No need to supply the cleanup procedures themselves. EPL generates these.

(c) No need to call the <unwinder> explicitly. The statement:

```
go to lab_2;
```

generates the call to the <unwinder>.

(d) No need to revert the "cleanup" handler immediately prior to the normal returns in <b> and <c>. EPL takes care of this by placing a suitable call to <reversion> in the so-called epilogue of the target code. See BP.3.00 for more details.

(e) If ON statements are used in place of calls to <condition>, even more service is provided by the EPL compiler. It will then automatically generate the calls to <reversion> immediately prior to the return statements.

Figure 5-12. Chain of Calls and Abnormal Return Coded in EPL

for inter-related procedures that include abnormal returns. The techniques employed by the EPL compiler to achieve this service for the EPL programmer are described in BP.3.00. Briefly, EPL generates the required calls to <condition> and to <reversion> for "cleanup" as required, i.e., for any procedure for which cleanup may be needed. It also generates and embeds the requisite cleanup procedure as an internal function. This function is referred to as the "epilogue" of the target code. The epilogue is ordinarily executed immediately prior to executing a normal return.

The clerical details performed by <unwinder> when it traverses a chain of calls that include ring crossings is more complicated than we have suggested in Figure 5-11. However the basic principal of frame-by-frame inspection, reversion, etc., is the same and, even more importantly, the net effect is the same. The details can be investigated in BD.9.05 by the stalwart. Whether you make this investigation or not, you should now be well convinced that a subsystem may be designed within Multics using or permitting others to employ abnormal returns, but the overhead for their oft-repeated use could prove to be prohibitive. Use of abnormal returns is in fact being avoided wherever possible in the implementation of Multics itself (Code value parameters are being used in place of statement label parameters).

Old MAD or FORTRAN lovers who are accustomed to using statement label parameters for abnormal returns should also be forewarned. Thus, if a compiler for MAD were to be implemented for use in Multics which accepted statement label arguments, one might well caution MAD programmers to restrict or to avoid use of abnormal returns, even if the compiler were to provide cleanup services similar to those now provided by EPL.