Published: 07/27/67

Identification

The Event Channel Manager

Michael J. Spier

Purpose

This section describes the Event Channel Manager which resides in the Administrative ring, and whose main functions are to create, maintain, delete, write into and read out of event channels. Special emphasis is put upon the relationship between two of the Event Channel Manager's modules, read event and set event, which work in conjunction with one another according to a special convention so as to permit interlock-free multiprocessor access to an event channel.

Note: It is essential to the understanding of this section that the reader be acquainted with MSPM Section BQ.6.03 which explains in detail all the terms used in the following discussion. Mention is made of the wait-coordinator, event-wait channels and event-call channels. These terms are associated with the reception of event signals and are explained in MSPM Section BQ.6.06. To briefly summarize, the wait coordinator is a procedure which is called by a receiving process when the latter can no longer continue its execution unless a specific event (or one out of a list of several events) has happened. The wait coordinator looks into the specified event channel to see whether that event has occurred. If yes, it returns to its caller otherwise it calls the block entry in the traffic controller. The receiving process will then wake up as soon as a sending process signals the event over the event channel. (It should be remembered that the signalling of an event signal over an event channel includes a call to wakeup for the receiving process.)

Introduction

As mentioned in MSPM section BQ.6.03, event channels may be simultaneously accessed by processes that are concurrently running on different processors. Event channels (and the two procedures responsible for writing and reading them) are defined in such a way that multiprocessor access to one single event channel can be carried out simultaneously. This interlock-free access is of interest not only because of the evident gain in time but mainly because of the fact that an event channel may be accessed by a ring 0 procedure that does not tolerate blocking for indefinite periods of time.

There are two different types of races between processes which access an event channel simultaneously:

a. A race between several sending processes. It is taken care of by the set_event procedure and (as will be seen) requires no special interlocking mechanism.

b. A race between a receiving process and one or more sending processes. This kind of race is taken care of by the read_event procedure and does require an interlocking mechanism, namely the switching of the two sub_channels.

The logic of the two sub channel systems is the following: If both sub channels are locked, the whole event channel is considered as being inaccessible. Both sub channels must <u>never</u> be <u>unlocked</u> simultaneously. Under normal conditions, one sub channel is always unlocked. A sending process may access such a sub channel provided that it take the following precautions (in the indicated order):

a. Ascertain that the sub channel is unlocked.
b. Increment the corresponding usage counter.
c. Verify that the interlock word has remained unchanged.

If conditions a and c are true, it may safely proceed to put its event indicator in the sub channel. If condition a is false it tries the second sub channel. If condition c is false, it means that the sub channel had in the meantime been locked by the receiving process, in between steps a and c. It decrements the usage count and tries the second sub channel, after having sent a wakeup signal to the receiving process.

A receiving process that wishes to read an event channel will always attempt to read the locked sub channel. If it finds it empty (reset) it will switch channels by switching the interlocking indicators in the interlock word. It then looks into the newly-locked sub channel to see whether the usage count is zero. If it is, it considers the sub channel to be safe for manipulating. If it is non-zero, this means that there is still some sending process working on that sub channel. The receiving process considers this condition to be the equivalent of a "signal not yet arrived" and will attempt to read the sub channel at a later time.

An event signal is read by decrementing the locked sub channel's event_count and (if it is an event-queue-mode channel), by detaching the first cell from the event-queue-list. A sub channel is considered to be reset if the event_count is equal to zero.

Following is a detailed discussion of set_event and read_event. The flow charts of both these procedures, figures 1 and 2, are provided as a reference.

Set event

A sending-process may send an event signal to a receiving process by calling the event channel manager entry set_event:

```
call ecm$set_event(rec_prcs,ev_chn,ev_id,sts)
```

```
declare (rec_prcs,sts) bit(36),(ev_chn,ev_id) bit(70);
```

where rec_prcs is a process id

ev_chn is an event channel name

(Note: It is assumed that initial (basic interprocess) communication has been established between the receiving and sending processes, and that the sending process knows both the receiving process' id and the event channel name. Basic interprocess communication is discussed in MSPM sections BQ.6.00-01.)

ev_id is an event identifier (see BQ.6.03), known to or generated by the sending process.

sts is a 36 bit string specified by the caller into which set_event puts return status information.

A sending process is supposed to know the process_id of a receiving process to which it wishes to send event signals. It may know it either implicitly or be informed of it by the receiving process via the basic interprocess communication mechanism. It may not use the Interprocess Communication Facility unless it knows both process id and event channel name. The event_channel_name will allow it to access the correct event channel within the receiving process' event channel table. The event channel table of a process is always known in that process' directory under the symbolic name <ect>. A process may access its own by using this symbolic name.

A sending process may access a receiving process' event channel table (provided that he have access rights to it) by using the path name

root > pdirdir > [receiving process id] > ect

When set_event tries to access, for the first time, a receiving process' event channel table or an event channel within that table, it knows them by symbolic name only (event channel table path-name, event channel name) and has to call upon modules which convert these symbolic names into machine-language pointers. Such conversions require considerable time, therefore they are done only for the very first access. The acquired pointers are then entered into an associative-memory type table (similar to the GE-645 associative memory register) which is consulted whenever set_event tries to access an event channel. This organization considerably accelerates the retrieval of a frequently accessed event channel. This speedup in access time is especially interesting to the I/O system which relies heavily upon the interprocess communication facility. (For details consult MSPM section BQ.6.09.)

The accessibility of an event channel to a sending process depends upon that process' access rights to the receiving process' event channel table as well as upon the sending process' and sending procedure's access rights to specific event channel within that table. Event channel protection is described in MSPM section BQ.6.05.

As described above, set_event looks at the sub channel interlock word. If both sub channels are locked the whole event channel is inaccessible to a sending process and an error return is made. When an accessible sub channel is found, set_event increments the usage counter by one, stores the event_id into the sub channel by a STAC instruction and increments the event count field with an add to memory type instruction (AOS). If the channel's signalling mode is the event queue mode, set_event reserves an empty cell in the Working Queue, writes the event indicator into the cell and then hooks the cell up to the end of the sub channel's event queue list.

This operation may be performed upon one sub channel by more than one sending process at the same time.

After having successfully put an event indicator into an event channel, set_event decrements the sub channel's usage_count. It then sends a wakeup signal to the receiving process and returns to its caller. The call wakeup(rec_prcs) is controlled by the event wakeup switch located in the Event Channel Table Header. This one-per-receiving-process switch is usually "ON", indicating that set_event should call wakeup for the receiving process.

When a process is quit by another process, the quitting process sets this switch to the "OFF" position, thus insuring that the quit process be not awakened by some sending process. Set_event does not call wakeup (receiving_process), when that switch is "OFF". This does not interfere with Interprocess Communication, it simply means that all arriving event indicators queue up in their respective event channels and are not immediately recognized by the receiving process. (See corresponding calls at the end of this section.)

We have described set_event functionally. Its implementation is somewhat more complicated due to reasons of protection and is discussed in MSPM section BQ.6.05.

Read event

The receiving process (when executing in the wait coordinator) reads the contents of an event channel by invoking the function

```
f=ecm$read_event(ev_chn,ev_ind,sts)
```

```
declare (ev_chn,ev_ind(3)) bit(70), sts bit(36);
```

where

- ev_chn is an event channel name
- ev_ind is a three element array, specified by the caller into which read_event puts the event indicator read out of the event channel. It includes:
  ev_ind(1)=event channel name
  ev_ind(2)=event id
  ev_ind(3)=sending process id (36 bit string).
- sts return status information.

The function f returns the value "1"b if an event indicator was successfully read out of an event channel, otherwise it returns the value "0"b. The status return reflects the specific reasons of failure for f="0"b.

We refer to the f="0"b condition as the "no results" condition. It is usually interpreted as implying "nobody signalled over this event channel".

Read_event is the event channel manager's counterpart to, and functionally associated with, the set_event procedure. the basic difference between them is that set_event may be directly accessible to any sending process' procedure (and even receiving process procedures) whereas read_event can be invoked by the receiving process' wait coordinator only. (See section BQ.6.06.)

Read_event looks into the locked sub channel. If there is information in it it proceeds to read it, otherwise it switches the sub channels' interlock status. If the newly locked sub channel is empty as well, a "no-results" return is made. Once that a non-empty locked sub-channel has been found, read_event checks the usage counter of that sub channel. Any non-zero value of that counter indicates that some sending process is still working on that channel, and is interpreted as a "signal not yet arrived" condition which causes a "no-results" return.

When read_event has found an accessible, non-zero sub channel, it reads <u>one</u> event_indicator by decrementing the event_count and by detaching the first cell of the event channel queue if the channel's mode is the event queue mode. When the event_count has been reset to zero, the channel is considered to be empty.

The information returned by read_event in the event indicator array depends upon the event channel's signalling mode:

event-count-mode (mode="0"b)	ev_ind(1)=event channel name
	ev_ind(2)=event id of the first event to have been signalled over the read sub-channel after the latter was last reset to zero.
	ev_ind(3)=0
event-queue-mode (mode="1"b)	ev_ind(1)=event channel name
	ev_ind(2)=this event's id
	ev_ind(3)=sending process id for this event.

Create ev chn

A receiving process creates a new event channel by calling:

```
call ecm$create_ev_chn(ev_chn,mode,signal_ring)
```

declare ev_chn bit(70), mode bit(1), signl_ring fixed bin(17);

where ev_chn is a location, specified by the caller, into which create_ev_chn puts the name of the newly created channel.

mode is the channel's signalling mode
"0"b=event-count-mode
"1"b=event-queue-mode

signl_ring is a ring number (0-63), specified by the caller, which corresponds to the lowest privilege ring from which a sending process' procedure may access this event channel.

The event channel thus created has the default attributes of a communication event-wait-channel. Access to it is restricted (by default) to member processes of the receiving process' group. (See MSPM section BQ.6.01.)

Declaration calls

The following calls provide the means of declaring the event channel's type attributes and of granting access to the channel to process-groups other than the receiving process' group.

an event-channel is declared to be an event-call channel by calling

```
ecm$decl_ev_call_chn(ev_chn,proc_ptr,data_ptr,prior,
    level,sts)
```

declare ev_chn bit(70), (proc_ptr,data_ptr) ptr, (prior,level) fixed bin(17), sts bit(36);

where ev_chn is the event channel's name

proc_ptr is a pointer to the associated procedure's entry point

data_ptr is a pointer to a data base associated with the event call

prior is a merge priority number which determines the place in the event-call-channel-list into which this channel is to be linked. The values assigned to prior are determined by the user who is expected to know what he is doing.

level is a level number used to inhibit recursive calls to the associated procedure from out of the wait coordinator. Any attempt to call the procedure recursively is validated by the wait coordinator only if the level number of the event-call-channel associated with the attempt is higher than the level number currently assigned to the associated procedure. Whenever a procedure is called by the wait coordinator, it gets assigned to it the level number of the associated event call channel. The value of level is determined by the user, as is the case for prior.

sts return status information.

Before accepting this call, decl_ev_call_chn checks with the file system to make sure that its caller is not trying to circumvent the system's protection mechanism, by declaring an event call channel so that he could indirectly call a procedure which is directly inaccessible to him. The call to decl_ev_chn is rejected if it turns out that ass_prcd may not be accessed from the caller's ring.

call ecm$decl_ev_wait_chn(ev_chn,sts)

declare ev_chn bit(70), sts bit(36);

where ev_chn is an event channel name

sts is return status information

disassociates an event-call-channel from its associated procedure and changes its type back to event-wait-channel. It does nothing if ev_chn points to an event-wait-channel, contrary to decl_ev_call_chn which always accepts legal calls, even for channels which already are declared to be event-call-channels, allowing channel reassociation to a different procedure.

Communication channels (which may be event-wait or event-call channels) are automatically coupled to device signal channels in the Device Signal Table when the user requests an I/O device. The channel is created in the normal manner, then associated with the I/O device by the Device Signal Channel Manager who puts a device index (the index of the device signal channel in the Device Signal Table) into the event channel header. The Device Signal Manager is described in MSPM section BQ.6.07.

```
     call ecm$give_access(ev_chn,acc_sw,acc_list)

declare ev_chn bit(70), acc_sw bit(1), acc_list(n)
         character(50);
```

where ev_chn is an event channel name

acc_sw is a switch which determines the procedure's function
"0"b=delete channel access list and make channel accessible to all (the acc_list argument is ignored)
"1"b=append acc_list to channel access list

acc_list is a list of 50-character process-group ids

grants access rights to all processes in the system or to member processes only of the process groups specified in acc_list, depending upon the state of acc_sw.

<u>Delet ev chn</u>

The call

```
     call ecm$delet_ev_chn(ev_chn,sts)

declare ev_chn bit(70), sts bit(36);
```

where ev_chn is an event channel name

sts is return status information

will have that channel deleted. The return status information will reflect the fact that the channel might have been deleted while still containing unread event indicators.

Calls associated with the wait coordinator

The declaration calls

call ecm$set_call_prior

call ecm$set_wait_prior

set the receiving process call-wait-priority switch (cl_wt_prior in the event channel table header) to "1"b or "0"b respectively.

The call to ecm$get_dev_signal(dev_signl_chn_list)

where dev_signl_chn_list is an index of the format fixed bin(17) which points to the beginning of the device-signal-channel-list

is made by the wait coordinator prior to any event channel interrogation in order to have the contents of the device-signal-channels in the device signal table transcribed into their corresponding associated event channels. (See MSPM section BQ.6.06.)

Calls associated with the quitting process

When a process quits another, it wants to set (and eventually reset) the quit process' wakeup switch.

call ecm$set_wakeup_sw(prcs_id,sw)

declare prcs_id bit(36), sw bit(1);

retrieves the target process' (prcs_id) event channel table, and sets the wakeup switch in that table to the value "sw".

call ecm$read_wakeup_sw(prcs_id,sw)

returns in "sw" the current value of the target process' wakeup switch.

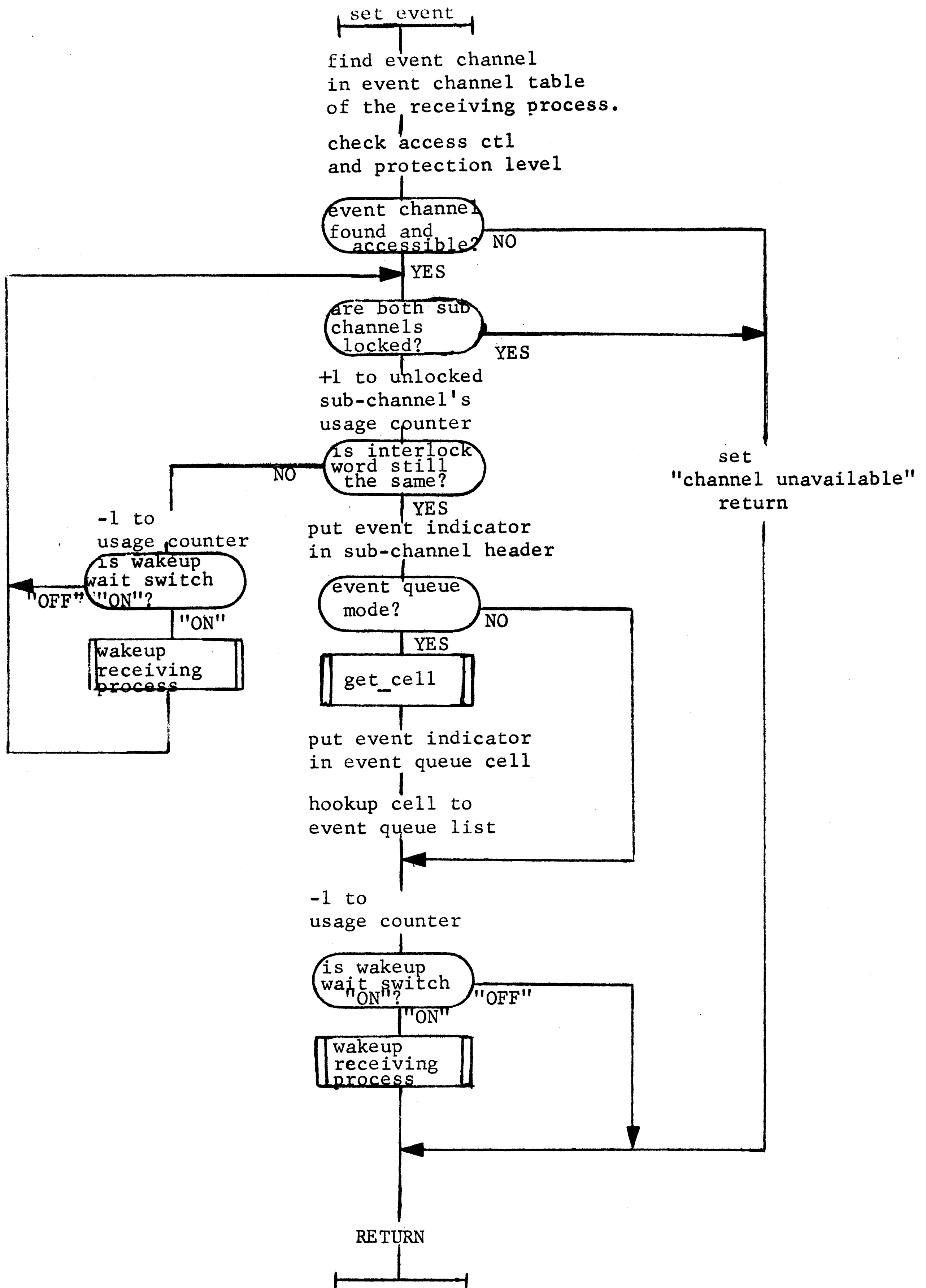

Figure 1: Set_event

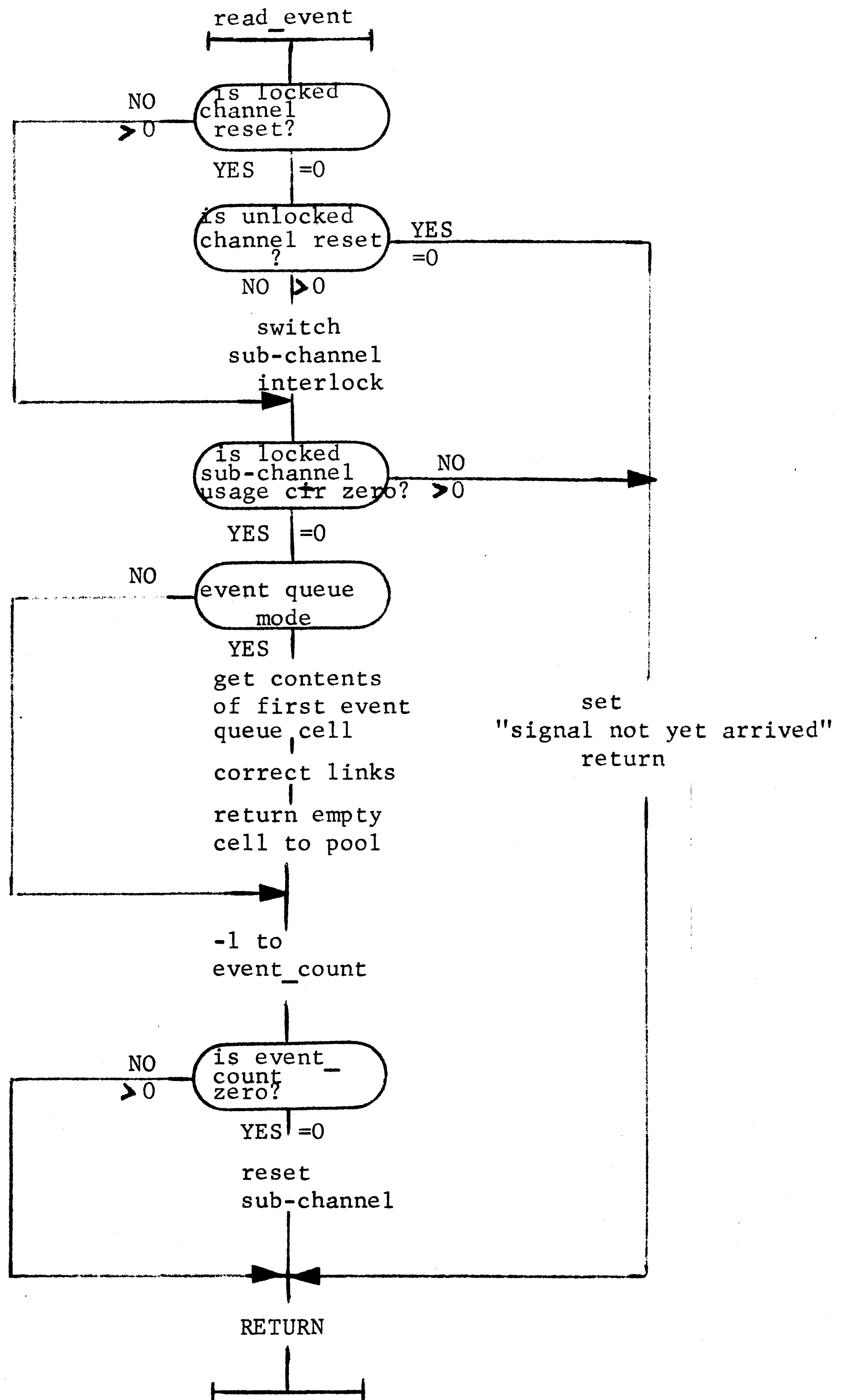

Figure 2: Read_event