TO: F.J. Corbato
J. Saltzer
A.C. Franklin
R. Freiburghouse
C. Garman
J. Gintell
N. Morris
R. Roach
M. Smith
V. Voydock
B. Wolman

FROM: J. Klensin
R. Sorrentino
(Cambridge Project)

DATE: Oct. 18, 1971

SUBJECT: New allocation and free procedures

Attached is draft documentation of area initialization, allocation and free procedures written for our use. As we have discussed with several of you, we believe that these procedures, or others much like them, should be incorporated into the system, and we are offering these for that purpose.

The following materials are included:

draft MPM documentation: area_
draft SWS documentation: alloc_, freen_
explanatory material on the purpose of these changes
logic descriptions of the routines

If desired, we will synthesize MSPM documentation from the logic material.

Proposed Modifications to MPM and SWS Documentation

Note that these changes, while adding clarity, do not affect the interface to these procedures in any way other than the addition of the new entry area_$no_alloc.

Entry: area_

The area_ procedure is called to initialize an area. This is a PL/I run-time library procedure which may be called by user programs.

Usage

```
declare area_ entry (fixed bin, pointer);
call area_ (size, area_ptr);
```

1) **size** is the number of words in the area. (Input)

2) **area_ptr** is an ITS pointer to the origin of the area. (Input)

Note

area_ sets up the area_ index after rounding the pointer to be equal to 0(mod 8). No check is made to assure that the size of the area is large enough to even hold the index.

Entry: **area_$redef**

This alternate entry is used to redefine the length of an area.

Usage

```
declare area_$redef entry (fixed bin, pointer);
call area_$redef (size, area_ptr);
```

1) **size** is the number of words that the area is to have. This size may be smaller than the original one. If so, the area which is to be released should not still be allocated, or an error of unknown flavor will result. (Input)

2) **area_ptr** points to the origin of the area. (Input)

Entry: area_$no_alloc

This entry point returns the number of allocations that have been made in the PL/I area pointed to by *area_ptr*. If the value returned is *zero*, then the area is *empty*, as defined by the PL/I language rules.

Usage:

```
dcl area_$no_alloc external entry (pointer) returns (fixed bin);
num = area_$no_alloc (area_ptr)
```

1) area_ptr is a pointer to the base of the area. (Input)

2) num is the number of allocations that currently exist in the area. (Output)

Name: alloc_

The alloc_ procedure is called by user programs to make an allocation in a PL/I area. It finds a contiguous block of words of a given size in a given area. This is a PL/I run-time library routine which may also be generally used.

Usage

```
declare alloc_ entry (fixed bin, ptr, ptr);

call alloc_ (size, areaptr, returnptr);
```

1) size — is the amount of storage to be allocated, in words. (Input)

2) areaptr — is a pointer to the base of the area in which the words are to be allocated. (Input)

3) returnptr — is a pointer to the first data word in the allocated block. This first data word will be on an even word boundary. (Output)

Notes: 1) The PL/I *area* condition will be signalled if there is insufficient space in the area in which to allocate the desired amount of storage. Upon return, the allocation will be retried.

2) The PL/I *error* condition will be signalled if the arguments to alloc_ are invalid, i.e. if *size* ≤ 0. Again, the allocation will be reattempted if returned to.

3) Only one process may modify an area an any given time. The condition *area_lock_wait* will be *signalled* if the area in which the allocation is to be made is and remains locked to the current process by another valid process. Upon return from an on-unit intercepting the signal, accessing of the area will be retried.

(END)

Name: freen_

The freen_ procedure is used to free a block of storage previously allocated in a PL/I area.

Usage

```
declare freen_ entry (ptr);

call freen_ (block_pointer);
```

1) block_pointer is a pointer to the base of the block to be returned to free storage. (Input)

Note

This is a PL/1 run-time routine which may also be generally used.

Only one process may modify an area at any given time. The condition *area_lock_wait* will be signalled if the area in which the allocation is to be made is and remains locked to the current process by another valid process. Upon return from an on-unit intercepting the signal, accessing of the area will be retried.

If block_pointer is invalid, or does not point to an allocated block in a PL/I area, the results of a call to freen_ are unpredictable.

(END)

Routines area_,alloc_,freen_
area_$redef
area_$no_alloc

Purpose To provide the dynamic storage management facility for the implementation of the based and controlled storage classes of the PL/I language, and also to protect the user's allocations from interrupts, and across processes and crashes.

Background After a study of the dynamic storage facility that was in use as of August 1971, it was decided that certain characteristics of those routines were, to varying extents, undesirable, and that other capabilities which were extremely desirable, were lacking.

These routines constitute part of the runtime library associated with the PL/I compiler, and as such, they must obey the language-specified rules for the handling of various errors which may be encountered. In particular, there are deviations from these rules which force the user to check for the occurrence of these errors in a non-standard fashion.

(1) If the allocation routine is called, and there is insufficent space available for the desired allocation, the *area* condition is not signalled as required by the PL/I language rules, but a *null* pointer is returned instead. Since this subroutine call to the dynamic storage manager is made implicitly by the PL/I *allocate* statement, it is a language violation to force the user to check the pointer which is returned.

(2) If arguments were passed to the routine packages which would cause undetermined action, but which were detectable, no attempt was made to notify the user via the PL/I *error* condition.

(3) The language provides that a return from an on-unit handli the area condition must be followed by an automatic re-try of operation. The organization of the old programs does not p this.

(4) The implementation of the *empty* built-in function, which is used, in PL/I, to set an area so as to contain no allocations, is difficult enough that this function has not been supported on Multics.

(5) In addition to the language problems, the routines themselves were coded in assembly language (ALM), and while this does not reduce their utility, it does complicate the maintainence of such programs over a protracted period of time. This is true in an environment where most of the interaction is done with programs coded in a higher level language (for the most part PL/I), particularly where much of the operating system is coded in such a manner.

(6) There exists no protection whatsoever against interrupts, nor operations attempted on the same area by two or more processes simultaneously. This also leaves no guarantee as to the state of the area should a crash occur while it is being modified. In a system, such as Multics, which deliberately interrupts a user after a pre-defined length of time in order to allow another user to resume execution, such protection borders on absolute necessity. It was suggested that some manner of stacking interrupts be employed so that whenever the area data base is examined (presumably to make a modification of some sort), it will always appear in a consistent, but possibly unpredictable state. This guarantees the usability of the database, after, perhaps, some small amount of cleaning up.

In addition, it is desirable to have some sort of locking mechanism which will cause one process to wait until another has completed any modifications it intends to make to the area. Finally, there should be preserved, in the area itself, sufficient information to allow its consistency to be restored automatically upon the next attempt modification of the area if a crash, or process termination, has left the area in an unusable state.

(7) The algorithm used to make allocations, and the corresponding data structure was also questioned. The one in use was the "buddy system", which subdivides the available storage into blocks which are powers of 2 in size. These blocks are paired, and as storage is freed, only if two "buddies" are free are they coalesced into a single block. If two adjacent, but "non-buddy" blocks are free, they are not merged.

Some research was done, and it was suggested that an algorithm related to that described by Knuth[1], a "first-fit method", employing a variation on the "boundary-tag system", would be used in place of the previous algorithm. This would allow for more efficient space management, and perhaps a reduction in the amount of time required for allocation.

All of the design goals described above were met. Some additional expense was incurred due to the increased complexity of the tasks to be performed. The size of the area header was increased from 22 words to about 80, to permit storage of the information required to maintain consistency across processes and system crashes. Further, although the time involved to make the allocation itself has decreased somewhat, an increase of 50-100 microseconds was imposed by the additional safety features, over the previous facility's total requirements. From preliminary timings, only 600-700 microseconds (no page faults, but including the 200 micro second PL/1 calling sequence) are required to perform the allocation. Freeing is somewhat less expensive, requiring 300-400 microseconds. All of these figures assume no page faults, no errors which will cause a PL/1 signal to be issued, nor any cleaning up to restore consistency before an allocation free operation is attempted.

---

1 The Art of Computer Programming, Knuth, Donald E., Volume 1, section 2.5. Addision-Wesley Publishing Company, copyright 1968

The procedures necessary to support PL/I allocation are as follows (names chosen from the current Multics implementation):

area_: Initializes area for allocations by other procedures. Used for implicit initialization and by the empty built-in function.

area_$ redef: Used to change the length of an area (amount of space available for allocations). This entry has no direct use in PL/I.

area_$no_alloc: Used to determine the number of existing allocations in and area. In particular, this entry may be used to determine whether an area contains any allocations. This function has no direct PL/I equivalent.

alloc_: Used to make an allocation of a specified length in a specified area. Corresponds to, and used by, the PL/I statement "allocate".

freen_: Used to free a specified allocation in a specified area. Corresponds to, and used by, the PL/I atatement "free".

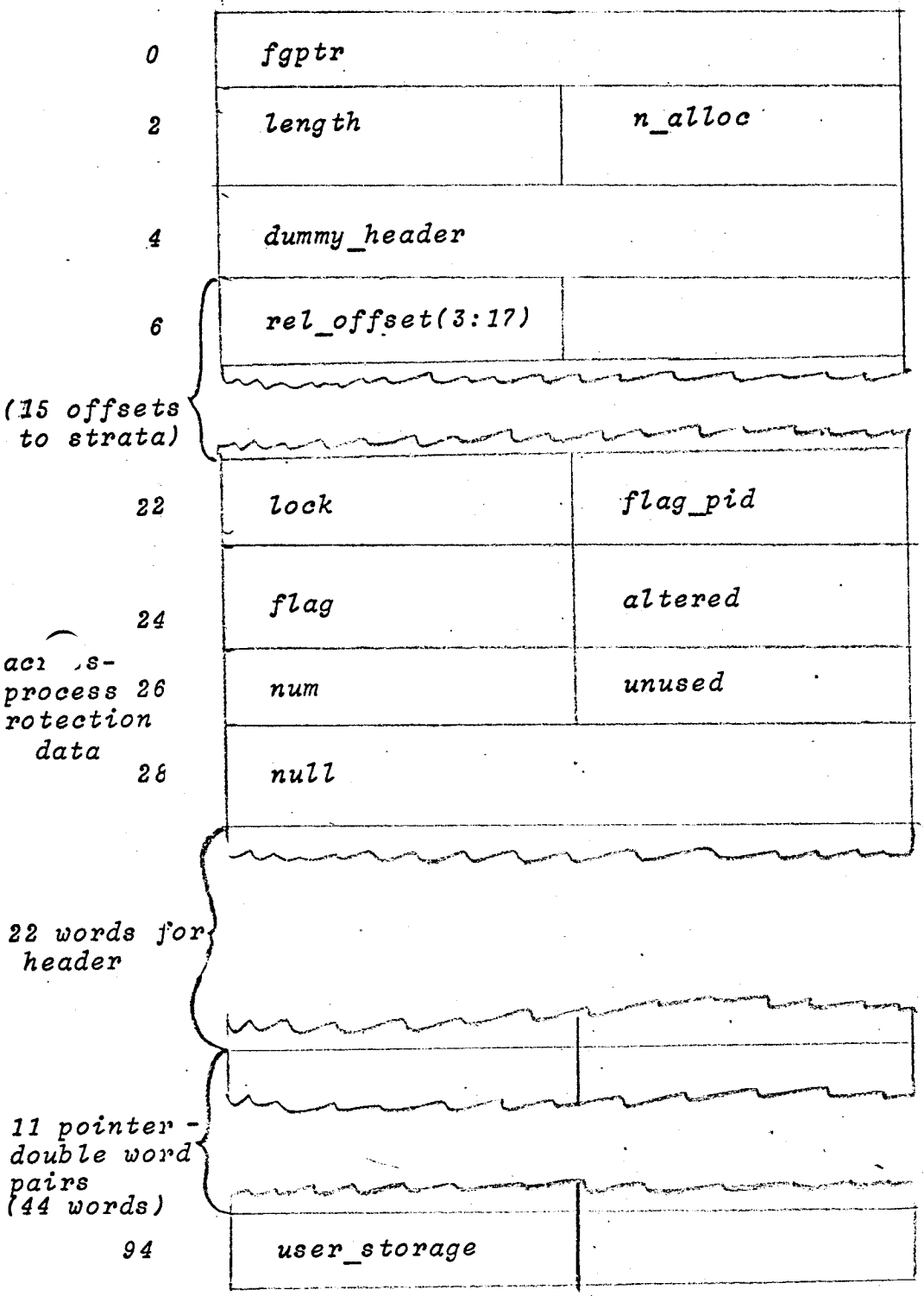
The remainder of this document details the algorithms for area initialization, allocating and freeing of storage, and for the interrupt protection strategy.

The data bases to which these descriptions make references are pictured on the next two pages. The algorithms (expecially the interrupt strategy) will be much easier to follow if one refers to these diagrams while reading.

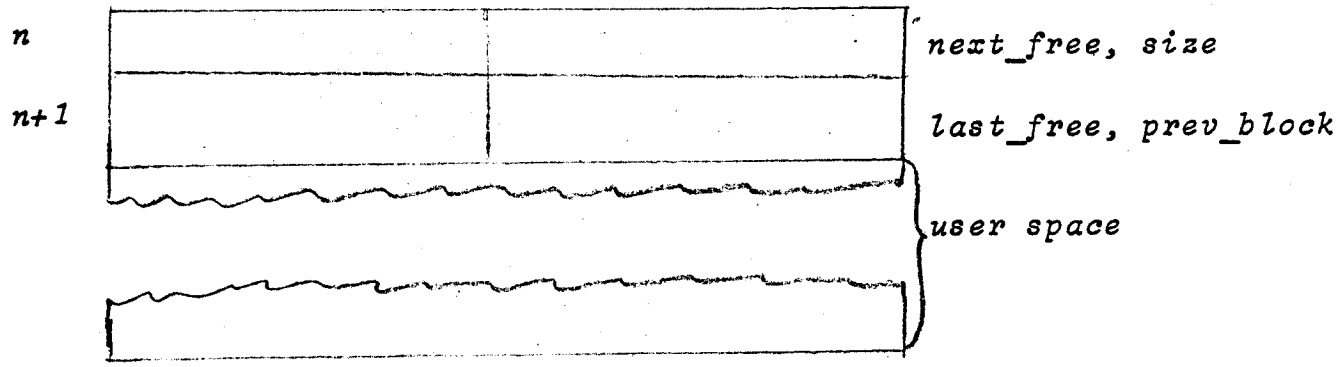
A "stratum" is defined as a chain of free blocks larger than a particular power of two. For example, a reference to "stratum(3)" implies that chain of blocks (chaining begins at the area header) larger than 2**3 words in size, but smaller than 2**4 words.

In order for the interrupt protection mechanism to work in complete safety, a hardcore entry will be necessary to set or clear a lock. This entry will be utilized only when an apparent conflicit occurs on a particular area. A description of the requirements for such an entry appears on the last page. This is not an attempt to exactly specify either the format or the name of this call.

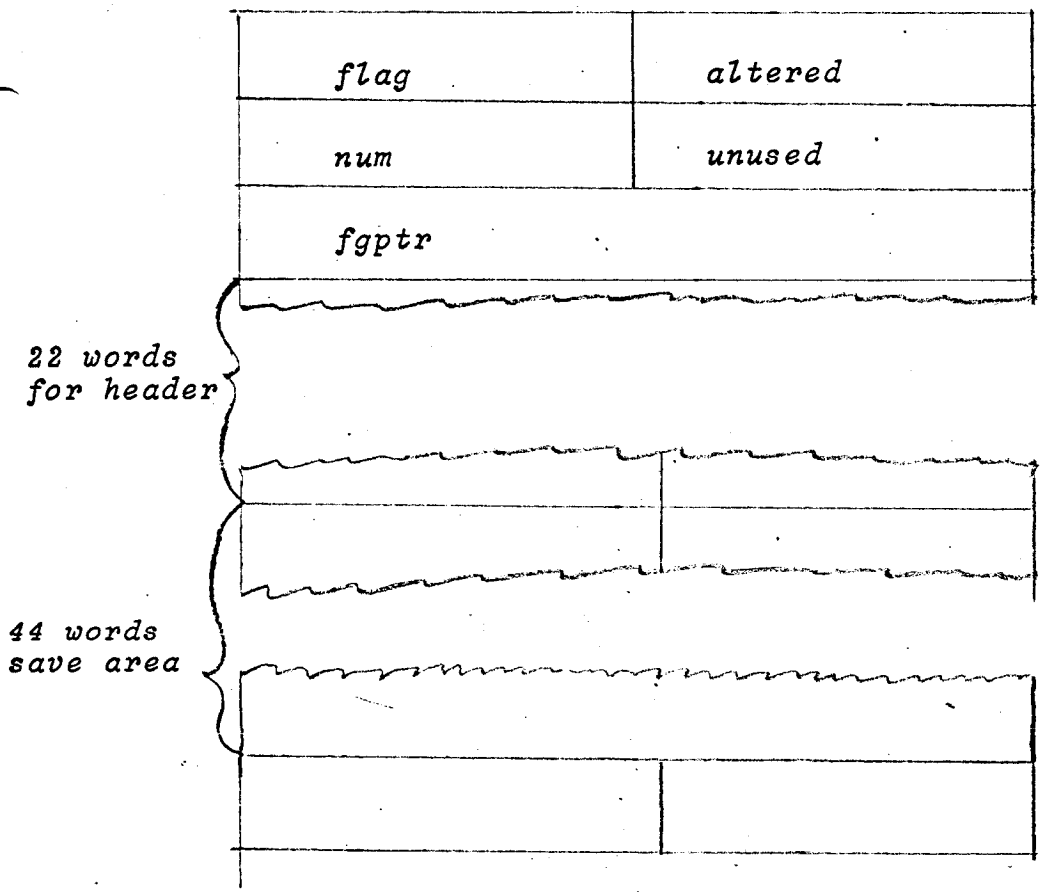
Header Structure



Block Structure



Communications Structure



AREA INITIALIZATION

Routine name: area_

(1) Round *area-ptr* to 0 mod 8, get desired area size

(2) Allot 92 words for the header, including save-area structure for across-process protection. Initialize the header.

(3) Compute the storage remaining for user allocations, and store it in *length*. Zero the stratum offsets

(4) Create a block header for the user space, and chain it into the appropriate stratum. If there is not enough space (< 2 words) after step 2, signal condition error. If returned to, go to step 2.

(5) Initialize *next_free* to "0"b, *last_free* to index of stratum in which it is chained, *size* to the number of words in this block (including the block header), and *prev_block* to zero, since there is no previous adjacent block which is free.

(6) Set up a dummy header for a non-existent block at the end of the area. This is used to keep track of the allocation status of the last real block in the area. Initialize *prev_block* to the offset (from area header) of the real user block.

(7) Return

BASIC ALLOCATION ALGORITHM

Routine name: alloc_

(1) Round area_ptr to 0 mod 8, obtain desired block size.

(2) Perform those checks necessary for protection against interrupts (described elsewhere). Assuming the area is in a consistent and usable state, these steps follow:

(3) Verify that the size desired is not negative. If it is signal *error*. On return, restart operation.

(4) Pad the request to an even number of words, and add 2 to the request to allow for the block header. Note that no less than 8 words is ever allocated.

(5) Determine the greatest power of 2 less than the desired size. Investigate this and all higher strata for the presence of a free block. In particular, since only the first member of a chain is examined, the size of the free block (in the entering stratum only) is compared with the desired size, that is, if it is too small, the search proceeds to the next higher stratum, not to the next block, if any, in the chain. If no block of sufficient size is found, the *area* condition is signalled. Upon return, the allocation is retried.

(6) If the found block is larger than the desired size by more than 8 words, it is split (Go to 7). Otherwise, go to 8.

(7) Determine the exact amount of extra space that is available. The new free block header will be located at addr (found_block)|n, where n is the desired size. Determine the stratum to which the unused portion of the split block belongs. If it is in the same one as the found block, reset the chains to be directed to it. Otherwise, reset the chains to exclude the found block, and chain the unused portion into the appropriate stratum. Fill in the headers of the allocated and unused portions, as appropriate. Go to 9.

(8) Reset the chains to exclude the allocated block.

(9) Set *next_free* to the offset from the area header. Indicate the state of the allocated block by setting the header of the block following it to zero in the *prev_block* field. If the block was split, update the prev_block field of the header of of the block following the unused portion to point to that portion, thus indicating that it is free for allocation. Increment the number of allocations, *n_alloc*, to account for this one. Zero the user portion of the block. Set *result_ptr* to addr (found_block)|2. Return to caller.

BASIC FREEING ALGORITHM

Routine name: freen_

(1) Obtain block pointer, and back up 2 words to the header. From the *next_free* field in the header, obtain a pointer to the area header. Obtain the size of the block.

(2) Examine the *prev_block* field to determine if the previous block is free. If it is, obtain a pointer, and add the size to that we already have. Flag that merging of that block is to be performed.
Repeat the procedure for the following block, by checking the *prev_block* field of the block following that one (the second block). In this way, we merge all adjacent free blocks into a single block.

(3) Determine the target stratum of the final block (after merging is done, if any). If that is identical to the stratum of the merged component (already free) in lowest position in the area, update the *size* field to indicate the new boundary of the merged block. If the block following that which we are now freeing was merged, unchain it from its stratum.
Go to 4.

(4) Set the *prev_block* field of the header in the block following the merged block to point back to that portion. Set the *size* field of the merged block as appropriate.
Decrement the number of allocations.

(5) Return to caller.

AREA PROTECTION

The following algorithm is observed during any freeing, or allocation operation, on an area. All changes which must be made to an area are first made in a double word (one for each block header in the area which is to be modified) in a save space, and a pointer to the actual location to be modified is saved as well. The area is actually updated as noted in the algorithm below.

(1) Check, and set to the user's process identifier, if zero the lock the area. If it was zero, go to 2, or if equal to the identifier of the current process, go to 3. Otherwise, call hcs_$clear_lock* to determine if the lock is set to the ID for a defunct process and set it to zero, if that was the case. If we have cycled through here more than 3 times, signal the condition "area_lock_wait". Upon return, zero the counter. In any case, proceed to 1.

(2) Check the flag in the save space in the area header. If it is equal to 2. the space contains valid data, and we must update the area. The double words are stored at the locations specified by the offsets in the pointers. In addition, the area header is updated. Verify that the contents of the save space were not altered. If they were, go to 2. Otherwise, go to 3.

(3) Check the flag pointer (area_header.fgptr) for a null value. If it is, go to 4. Otherwise, compare the process id associated with that pointer with the id of the current process. If it is not equal, set the pointer *null*, and *flag_pid* to the current process id. In any event, go to 4.

(4) Save the current flag pointer in the local automatic storage save space. If that pointer was null, go to 5. Otherwise, check the flag for a value of 2. If it is, perform the following: {Otherwise, set that flag equal to 1.}

(A) Copy the updated version of the area header to the real location. If our data was altered ("altered bit" on) while we were storing, go to A.

(B) Similarly, store the double words at the locations specified by the associated pointers. If the data was altered, go to B.

Note that step C. is performed only if the good save space belongs to this level. Otherwise, go to 5.

(C) Chain backwards through all flag pointers until a null pointer is found. At each level, mark the data *altered*, and wherever a target location is found to be the same as one at the topmost level, change the value of the corresponding double work at that level to that at the topmost level. This insures that all

* See Attached

levels will update again with the correct information. If we were interrupted, and our data altered, go to C. Back up the area header flag pointer to the previous level, if any. Go back to point of invocation.

(5) Set the flag in the local automatic save space to zero, and point the area flag pointer to the save space. The zero value indicates to higher levels that we are merely gathering data for our operation. When we have finished such accumulation, we will check the flag, and if it is zero, we set it to 2. If it is equal to 1, we were interrupted, and must assume that the gathered data is invalid, and we must begin the procedure again. The check of the flag is performed via the *stac* (STORE ACCUMULATOR CONDITIONAL) instruction, which is non-interruptable. This is implemented in Multics PL/1 as the *stac* built-in function.

(6) Perform all data operations upon the save space, as appropriate to allocation or freeing. (Described in those sections).

(7) *stac* the flag in our save space, as described above.

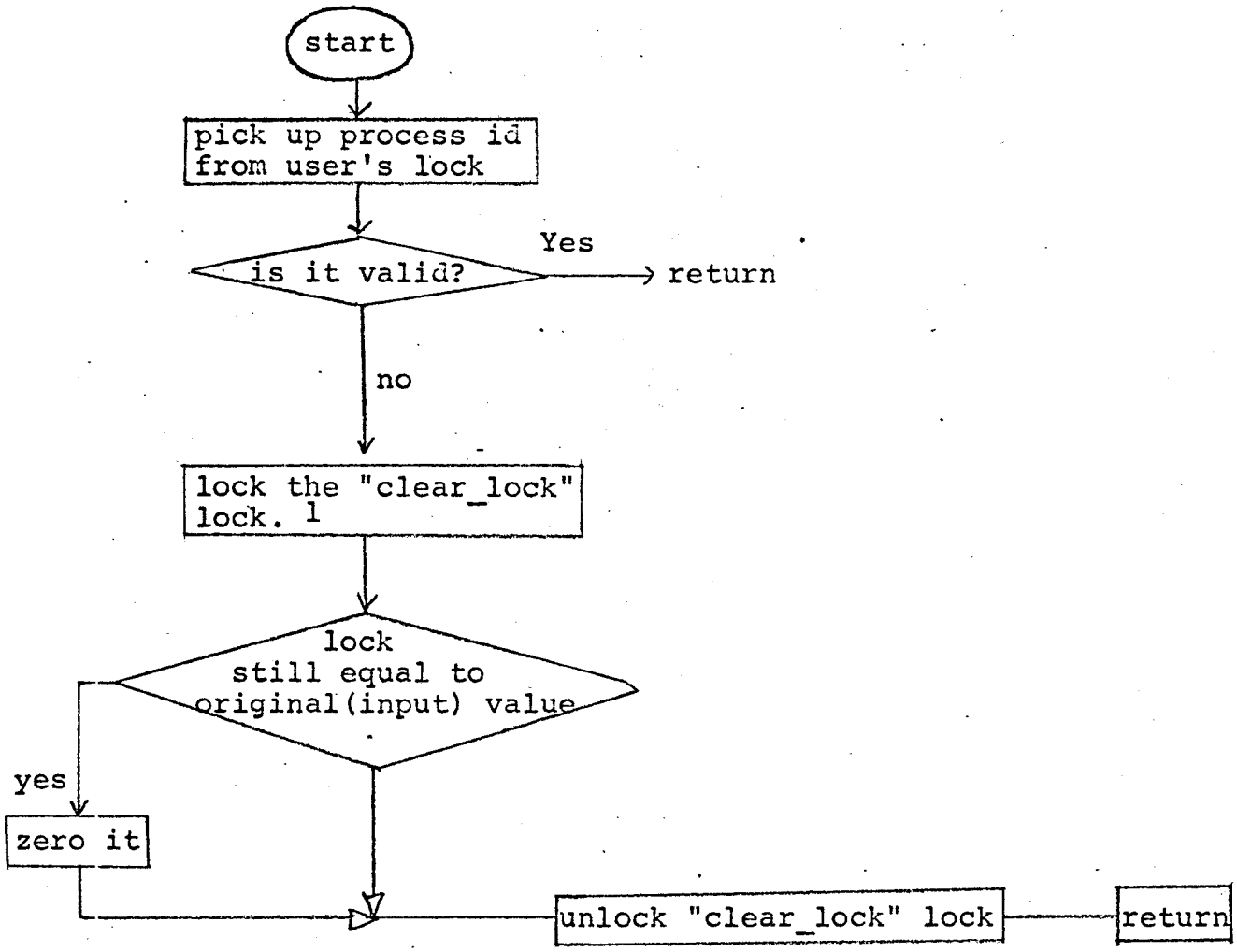
(7A) If the *stac* operation was successful, mark the static (in area) save space as "no good" (raise a flag), and update it from our automatic save space. Verify that our save space did not change, if it did, go to 7A. Otherwise, mark the static space as "good".

(8) Perform the operations described in sections 4A-C. Complete allocation or freeing operation, and return.

Minimum Algorithm Required for Supervisor clear_lock entry

```
declare hcs_$clear_lock entry (pointer);
call    hcs_$clear_lock (area_lock_ptr);
```

This procedure must validate the 36 bit process id pointed to by area_lock_ptr. If it does not belong to an existing process, the lock is to be set to zero.



---

1 This lock is used to prevent more than one process from executing this code at once. It is a per-system lock cleared by the Multics salvager.