Published: 07/24/68

Identification

Hardware Configuration Checker
A. Sasaki

Purpose

Hardware Configuration Checker is a subroutine which can be called immediately after the initialization of the MMCT (BK.4.04) in the system initialization environment to check (so far as possible) that the hardware configuration information in the MMCT agrees with the actual hardware modules physically existing in the system and the actual switch settings or plug boards used on those modules.

Calling Sequence

```
call check_configuration (status);

dcl  status fixed bin (36);     /*If status = 0 upon return
                                  then check_configuration
                                  believes that the MMCT
                                  agrees with the actual
                                  configuration. It is
                                  possible, however, that
                                  they disagree even if
                                  check_configuration believes
                                  that there is no problem
                                  because many switches and
                                  plug boards are totally
                                  program inaccessible. If
                                  check_configuration detects
                                  any problem then "status"
                                  is set to a non-zero value.
                                  As to the assignment of
                                  values to the "status" see
                                  Implementation*/.
```

Discussion

Generally, it is impossible for a program to directly sense the hardware switches on the GE645. Instead, indirect but persuasive checks on the MMCT entries must be devised. Another general principle of check_configuration is that it does NOT attempt to check absense of anything. Check_configuration operates in the simple EPL environment provided by the bootstrap I, and II (see BL.0, BL.7.00) and the following tests are performed in the order of the description.

1. Tests on Memory Controllers for address ranges.

2. Tests on inter-module cable connections.

3. Tests on CPU's for proper base address, controller assignment, and CPU tags.

4. Tests on GIOC's for proper base address, response, and interrupt cell assignment.

5. Tests on Drum Controllers for proper base address, ability to read data out of the drum, and interrupt cell assignment.

6. Tests on System Clocks for proper address, reasonable setting, counting, interrupt cell assignment, and discrepancies between them.

All tests involve hardware-software interactions in their clumsy and unembellished forms in a multi-processor hardware environment and in the simple EPL environment, and thus all tests are destined to be acrobatic if not tricky. When execution of check_configuration begins, only the bootload processor is guaranteed to be sane and all of non-bootload processors have psychoses. Namely, at first all registers on non-bootload processors contain random numbers and the processors are in DIS state. Therefore non-bootload processors are presumed to be insane and as dangerous as psychotics with sharp cutleries in their hands at first. Once they are wakened up from DIS state by being interrupted or faulted they are quite ready to start destroying everything and what is more, a number of them can do that concurrently. Non-bootload processors are not totally insane, however, since they may be brute forced to execute an XED instruction in the area specified by settings of the base address switches upon being interrupted or faulted. See BK.1.04. In Multics system, the base address switches are set in the same way on all processors by convention. Therefore, the only sane processor, the bootload processor, does know where those insane processors are supposed to report temporarily sane when interrupted and faulted, and he can put goodies there (accessible as <fault_vector>) to let the insanes do a tiny bit of sensible thing for the purpose of the tests, say painting their own faces (loading its own dbr etc.).

Various tests on GIOC's and Drum Controllers can be devised based on the above philosophy by attempting to force the GIOC or Drum Controller under test to cause a particular interrupt of interest to check_configuration.

It is noted that any processor sets interrupt cells on any Memory Controller, but any particular GIOC or Drum Controller sets only interrupt cells on his mother Memory Controller who in turn reports to his mother processor who is not necessarily the bootload processor and could well be one of the psychotics.

For testing interrupt cell assignment (BC.1.04), it is necessary to know which interrupt cell was actually set by the device under test. Unfortunately, however, interrupt cells are not directly program accessible and an awkward and indirect procedure is mandatory. Namely, the bootload processor identifies the interrupt cell set on a Memory Controller by letting the interrupted processor leave a trace of different actions corresponding to each different interrupt word pair in some area symbolically accessible for the bootload processor.

The following are future elaborations foreseeable at this writing, but not included in initial implementation:

1. Selective masking could be used to positively identify the Memory Controller on which the interrupt cell was set.

2. Checking for inconsistencies among MMCT entries themselves before starting the tests listed previously would be of great value.

3. With simple I/O control program designed just for the purpose of tests, it should be possible to test full internal configuration of the GIOC under test, and Drum Controllers could be tested for more elaborate operation.

4. It should be possible to test the associative memory switch on the CPU panel, either by comparing the execution times for a trial access to some location preceded and not preceded by a CAM instruction; or by a combination of a SAM instruction and a trial access to some location. Since SAM stores the usage count bits of the associative memory words it is possible to positively identify the setting of the switch by checking the bits before and after the trial access.

5. The timer register switch could be tested by executing a STT instruction before and after carrying out trial accesses to some location several times. If the switch is set to memory cycle the timer register will be decremented by the number of the trail accesses. If the switch is set the other way the register will not be proportionally decremented since the period of the 64Kc clock is about 15 microseconds.

Implementation

Check-configuration is implemented as an EPL procedure that calls other EPL subroutines each of which in turn calls one to several very small hand written EPLBSA utility routines designed for execution of special instructions. Check-configuration checks the return argument "error" each time upon return from the subroutines and if it has been set to a non-zero value he passes that value to his return argument "status" and returns to the <initializer>, otherwise he continues the rest of the tests. Overall implementation and details of the EPL and EPLBSA subroutines are shown in the following.

```
check_configuration: procedure (status);

call test_memory_controller (error);
                        /*error = 10 + i means that expected
                          addresses do not exist on the i-th
                          memory controller where i is the
                          MMCT index,
                         error = 19 means other troubles*/

call test_control_assign_and_wiring (error);
                        /*error = 20 + i means that a SMIC
                          or CIOC related to the i-th memory
                          controller has resulted in an
                          anomaly where i is the MMCT index,
                         error = 29 means other troubles*/

call test_cpu (error);
                        /*error = 30 + i: a trouble on the
                          i-th cpu in the 1st part of
                          test-cpu;
                         error = 40 + i: a trouble on the
                          i-th cpu in the 2nd part of test-
                          cpu, where i is the MMCT index,
                         error = 39 or 49 means other
                          troubles*/

call test_gioc (error);
                        /*error = 50 + i: a trouble on the
                          i-th GIOC where i is the MMCT
                          index,
                         error = 59 means other troubles*/

call test_drum_controller (error);
                        /*error = 60 + i: a trouble on the
                          i-th drum controller where i is
                          the MMCT index,
                         error = 69 means other troubles*/
```

```
call test_system_clock (error);
                    /*error = 71: the primary clock is improperly
                                  addressed.
                      error = 72: a secondary clock is improperly
                                  addressed.
                      error = 73: the primary clock has not been
                                  set properly.
                      error = 74: a secondary clock has not been
                                  set properly.
                      error = 75: the primary clock is not
                                  running.
                      error = 76: a secondary clock is not
                                  running.
                      error = 77: the primary clock has wrong
                                  interrupt cell assignment.
                      error = 78: a secondary clock has wrong
                                  interrupt cell assignment.
                      error = 79: the discrepancy between clocks
                                  is greater than 1 sec.*/
```

test_memory_controller

The core memory existing in the system is divided into independent modules each controlled by an independent Memory Controller. Each Memory Controller is assigned to a distinct address range. The address range assignment specified in MMCT is checked by:

1. Setting the memory out of bound fault to transfer the control check_configuration (see Fig. 1).

2. Attempting to read the highest and the lowest addresses in each Memory Controller, as specified in MMCT.

The highest and the lowest addresses in a given address range can be generated by the same technique as is used in the <scas_init>. See BC.3.04, BK.4.02, BL.2.01, BL.2.02. If no fault occurs, the claimed memory assignments do exist somewhere although not necessarily in the Memory Controller expected.

Preceding the tests on the Memory Controllers, the <fault_vector> is saved (see BK.1.04). A dummy segment is used (see Fig. 2) and its DSW is manipulated to let it point to the highest and the lowest addresses in the address ranges. Figure 1 shows the temporary <fault_vector> and EPLBSA subroutines used for the purpose of the trial readings. The ITS pairs in the <fault-vector> and the <touch_snapshot> must be initialized before the tests. It is noted that when control is transferred to the <touch_snapshot>, the ABR's are still loaded with quantities relevant to the <read_dummy_seg> Therefore all external reference within the <touch_snapshot>, must be done via ITS pairs in order to avoid relying on the

linkage mechanism. The <touch_snapshot> increments the stored ICTC content by 1 when the stored CPU tag agrees with the bootload CPU tag in the MMCT. It is test_memory_controller's responsibility to check the validity of the tag in the MMCT before starting the trial readings by interrupting the bootload CPU calling set_cell (see Fig. 2). Thus, when the bootload CPU is faulted, it will continue the next step of the tests and if it is a non-bootload CPU, it will go back to the DIS state. In case any of the CPU's is interrupted, the snapshot is stored and the CPU will either go back to the DIS state or continue the next step of the tests as in the case of the faults.

While any non-bootload processor is executing in the <fault_vector> or <touch_snapshot>, it runs concurrently with the bootload processor. Therefore, the bootload processor must wait for completion of other processor's action each time after its trial read-out by looping in EPL.

test_control_assign_and_wiring

Check-configuration executes CIOC's and SMIC's on the bootload processor for testing inter-module cable connections and control processor assignment. The CIOC's are used to connect-fault the aimed processor. The <fault_vector> in the Figure 1 is used for this test also. If both the cable connection and control processor assignment are correct, then either the bootload or a non-bootload processor will leave the CPU snapshot at <fault_vector>|576. If MMCT entries for the control processor assignment is in error, however, a different processor than described as the control processor in the MMCT will be faulted at best, and even a non-CPU device could receive the operand word of the CIOC. If an error in the cable connection and control assignment happen to exist in a particular combination, it is possible that the seemingly correct CPU tag does not mean correct configuration; correct CPU tag could mean wrong control assignment and wrong cable connection. This is the reason why tests by CIOC's as well as tests by SMIC are needed. If a correct CPU picture is found a short while after executing a CIOC, that means the processor is physically connected to the specified part on the specified Memory Controller, and the possibility for the double error in the tests using SMIC's can be forgotten. Since, if switches are set wrong, the operand word of the CIOC can be transmitted to any device, the operand word must be fabricated so as NOT to cause disastrous uncontrolled run of any device. A possible bit pattern that can be used as the CIOC operand is

"000000000000000000000000000000101XXX"b

On a processor, the bit pattern is immaterial, and on a GIOC the bit pattern is interpreted as "connect-channel-9 for routine work should be activated in the normal mode". Thus, by specifying the IPW tally as "0"b preceding the test, it is possible to let the GIOC interrupt a particular CPU. In other words, if a fault has occurred that means a CPU has been connected and if it is an interrupt that means a GIOC has been connected. If nothing has occurred after having executed a CIOC then something is wrong or a Drum Controller has received the CIOC operand. On a Drum Controller, the bit pattern is taken for a PCW specifying "Emergency Disconnect" (see Fig. 5). When execution of check-configuration begins, all relevant segments have been loaded and prelinked (see BL.7.00), and thus the drum should NOT be in busy status and the "Emergency Disconnect" will be disregarded. Since the CIOC instruction does use the operand word, it is necessary to store the operand at some location. It is a fundamental hypothesis to the initialization procedure that at the end of Collection 1 the size of the whole system should NOT overflow the first memory module. Therefore, any core location in the second and up core blocks can be used for storing the CIOC operand at system writer's own cost of generating absolute addresses pointing to outside the first memory module, and the CIOC operand can be stored at any location within the program if the CIOC is to be directed to the Memory Controller for the first memory module.

test_cpu

In the first part of the tests on CPU's, the content of the dbr on the bootload processor is copied into dbr's on non-bootload processors, and a new version of the <fault_vector> is used for this purpose. The interrupt vectors are set to transfer control to a pair of ldbr instructions and an indirect transfer using the same technique as for processor initialization (see BL.11.03). The second part of the tests on CPU's consists of procedures for fabricating still another version of temporary <fault_vector>, and procedures for testing CPU's. In the "realistic" environment, the non-bootload processors are not insane any more but they are still imbeciles since their address base registers have not been initialized or paired yet. Thus, they can access those locations pointed to by a given ITS pair because of the proper content of their dbr but such medium IQ actions as using the stack, or linkage section are still beyond their ability. The <fault-vector> used in the second part allows check_configuration to identify the interrupt cell set on a Memory Controller.

First Part:

The new <fault_vector> used in the first part and other relevant segments are shown in the flow chart, Figure 2. The XXX----X in Figure 2 is the content of the dbr on the bootload processor, and the <dummy_seg> is a dummy segment used as the operand address of the SMIC's. The address of the <dummy_seg> is properly modified preceding execution of each SMIC for interrupting the particular CPU to let it load its own dbr. At the end of the first part, all of the non-bootload processors are in DIS state with their dbr's containing the same content as the dbr on the bootload processor who is ready to initiate the second part.

Second Part:

Still another version of <fault_vector> is fabricated at the beginning of the second part. The layout of the new <fault_vector> and whole picture of the second part is shown in Figure 3. Check_configuration sets a different interrupt cell in each Memory Controller, one by one, for the purpose of confirming that the new <fault_vector> actually allows it to identify the cell set on a Memory Controller.

The fourth word of the control unit information is always non-zero. Thus, if 6*(i-1) + 4 th word of the <snapshots> is found non-zero that means the i-th bit of some interrupt cell has been set. Now check_configuration checks the tag bits of the stored control information and identifies which processor has been actually interrupted. At the end of the second part, it is highly persuasive that base address of the CPU's are properly set, controller assignment is correct, CPU's are properly tagged, and no processor is psychotic if not smart. The environment established by the second part will be used in the subsequent steps of the check_configuration.

test_gioc

The basic tactic for tests on the GIOC's is to issue a connect, setting the tally field of the IPWB to "0"b preceding the test. See Figure 4. When a GIOC has been connected, it checks that part of the IPWB before it starts any task and if it finds that portion "0"b the status channel 0 returns an emergency status word and sets an interrupt. (N.B.: audible alarm is not triggered.) The spare words of the mail box of the GIOC under test will be used for storing both the emergency status word and the CIOC operand.

A pause in EPL is necessary to wait for completion of storing the emergency status word and setting the interrupt cell. Then check_configuration goes to the stored information and checks it for expected emergency status word and control unit information at expected locations. If the check indicates no anomaly, it is highly persuasive that the GIOC under test does exist actively responding with proper base address and interrupt cell assignment in agreement with the MMCT entries.

test_drum_controller

The basic tactic for testings on Drum Controllers is to carry out CIOC's directed toward the particular Drum Controller under test with the operand word fabricated for reading out a 64 word page. The hardware design of Drum Controller requires that the CIOC operand must be stored in a location on the memory module where the mail box of the Drum Controller is maintained. See figure 6. Using the 32nd word for this purpose should not cause any trouble since check_ configuration is the only running process and all status words should have been processed when execution of test_drum_ controller begins. The PCW stored at the 32nd word of the mail box has command bits for "start, fetch DCW pair". The 34th and 35th words are used for storing the DCW specifying "Read out a small page and set interrupt cell", and the 36th and 37th words are used for the DCW specifying "Normal Disconnect". A pause by looping in EPL is necessary to wait for completion of the data transfer, disconnect and the interrupt handling when the CIOC is executed. The <fault_vector> is Figure 3 is used since checking the interrupt cell assignment is one of the requirements. If data has been read out to the core area reserved for the trial reading, and if the expected control unit information is found at the expected location as well, then it is highly persuasive that the Drum Controller under test does exist actively responding with proper base address and interrupt cell assignment in agreement with the MMCT entries.

test_system_clock

The check_configuration reads all of the system clocks sequentially one by one, and checks the readings against "0"b, and "111----11"b then compares the readings. If any of the readings of the secondary clocks disagree with the primary clock by more than one second, check_configuration picks up the largest of the readings and add 10 microseconds to

it. Then it waits for one second by looping in EPL and sets the alarm clock registers to that value one by one. If the clock is counting properly, a wake up interrupt will occur immediately. The <fault_vector> in figure 3 is used and the interrupt cell set by the wake up interrupt is positively identified for the purpose of checking the interrupt cell assignment. It is necessary to read secondary clocks by an EPLBSA subroutine using the segment <clock_>. The tests outlined here check all of the system clocks for proper address, reasonable setting, counting, and correct interrupt cell assignment for the wake up interrupt. (The "trouble interrupt" cell assignment is not checkable by program.)

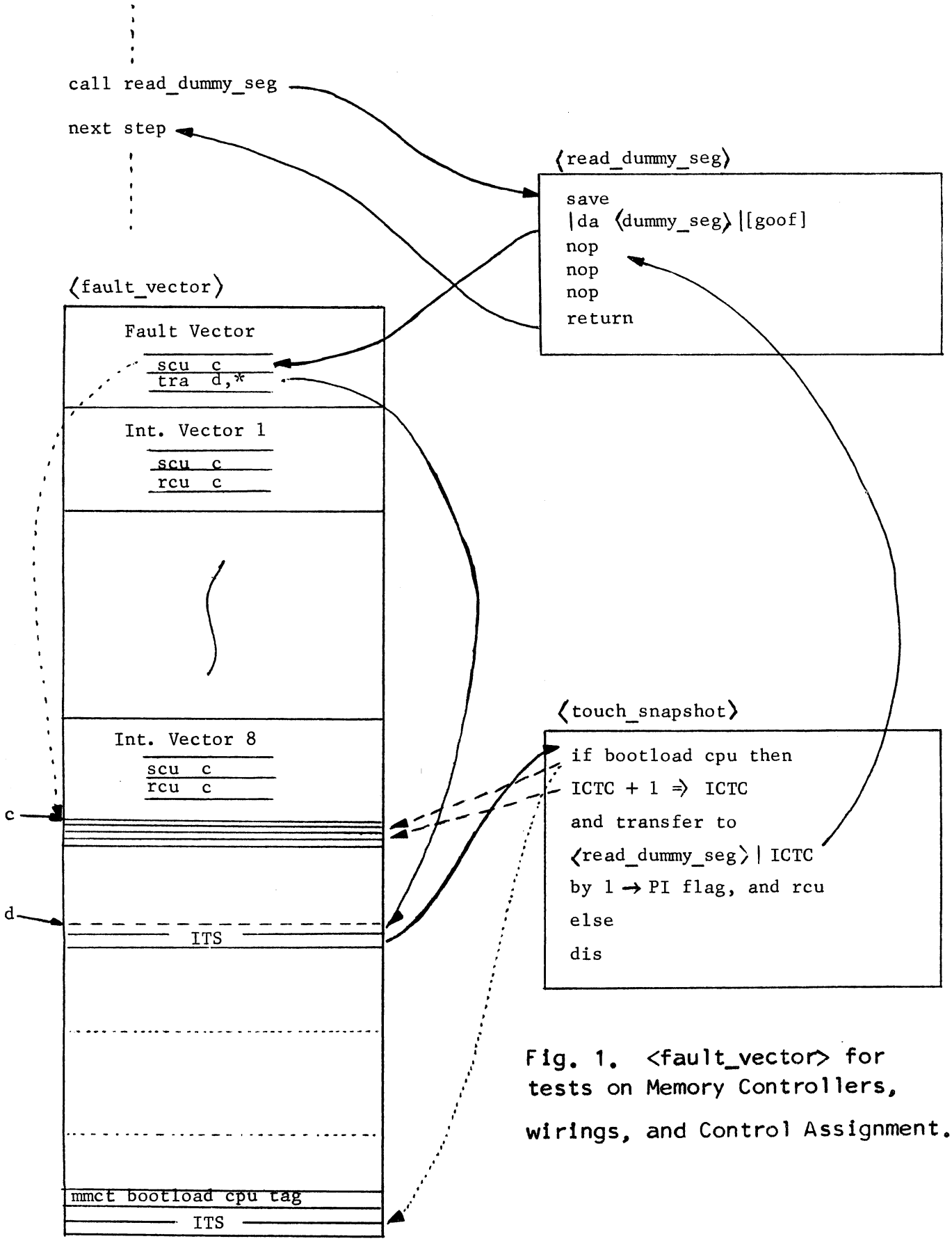

Fig. 1. <fault_vector> for tests on Memory Controllers, wirings, and Control Assignment.

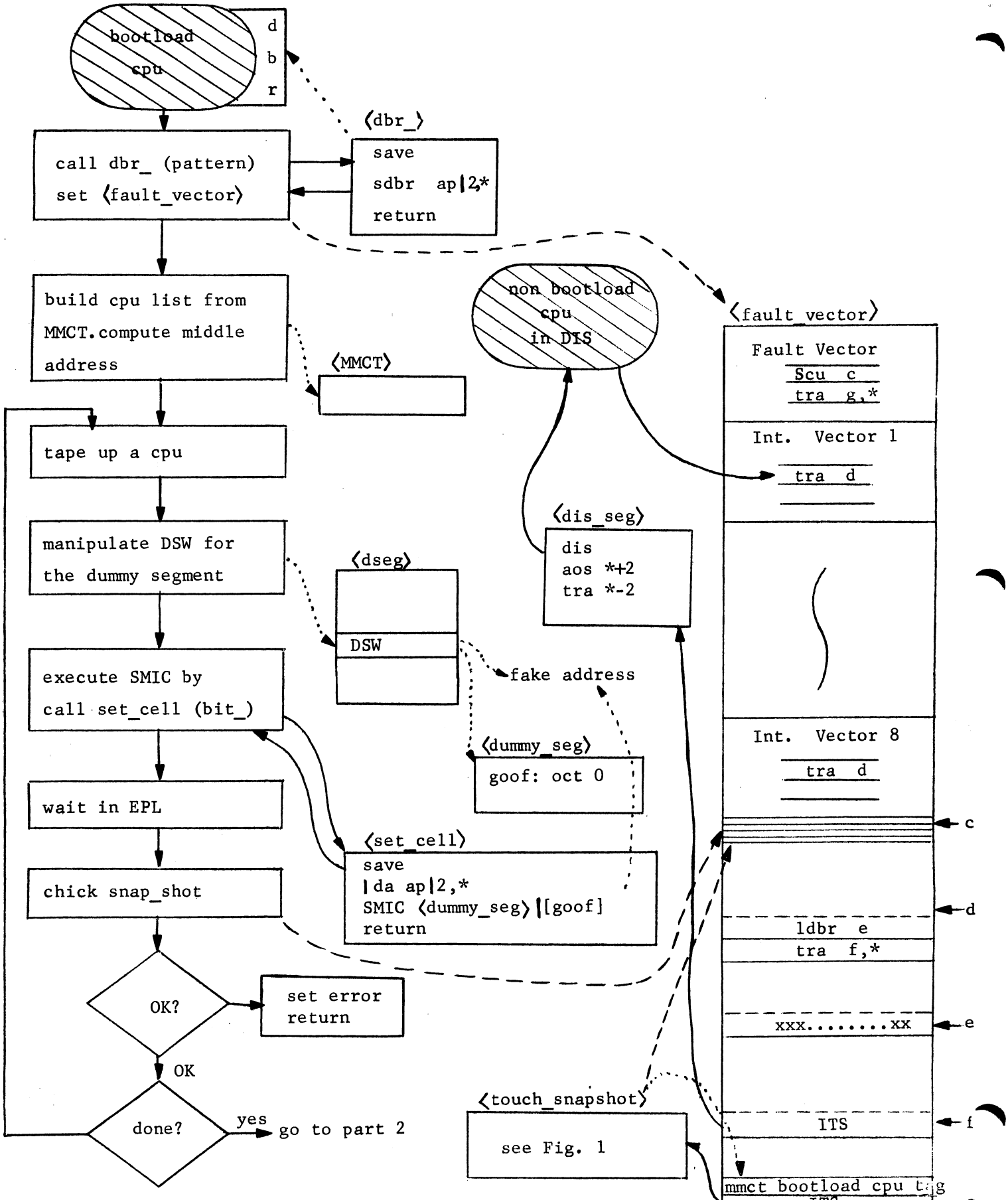

Fig. 2. CPU test part 1

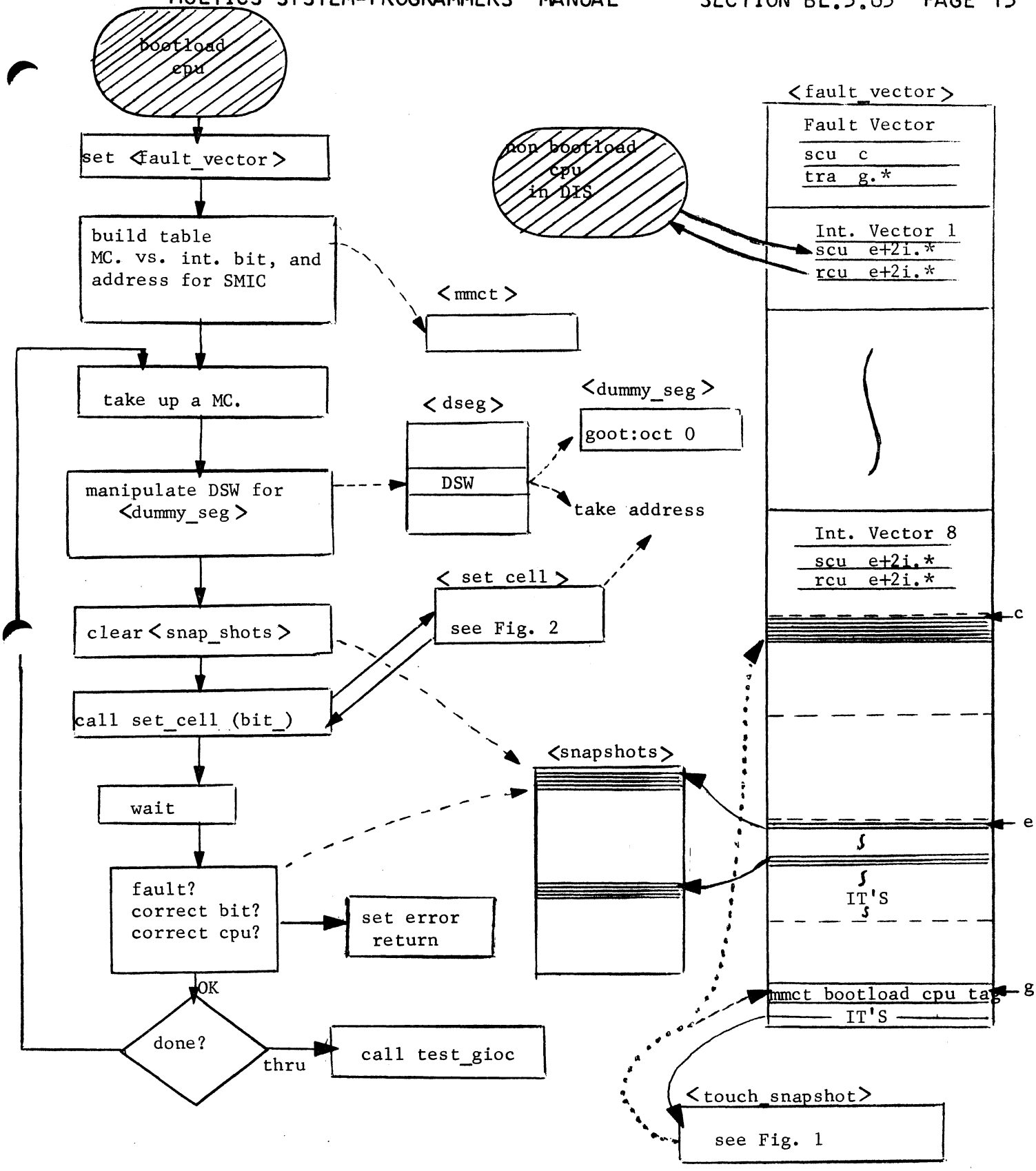

Fig. 3: CPU Test Part 2.

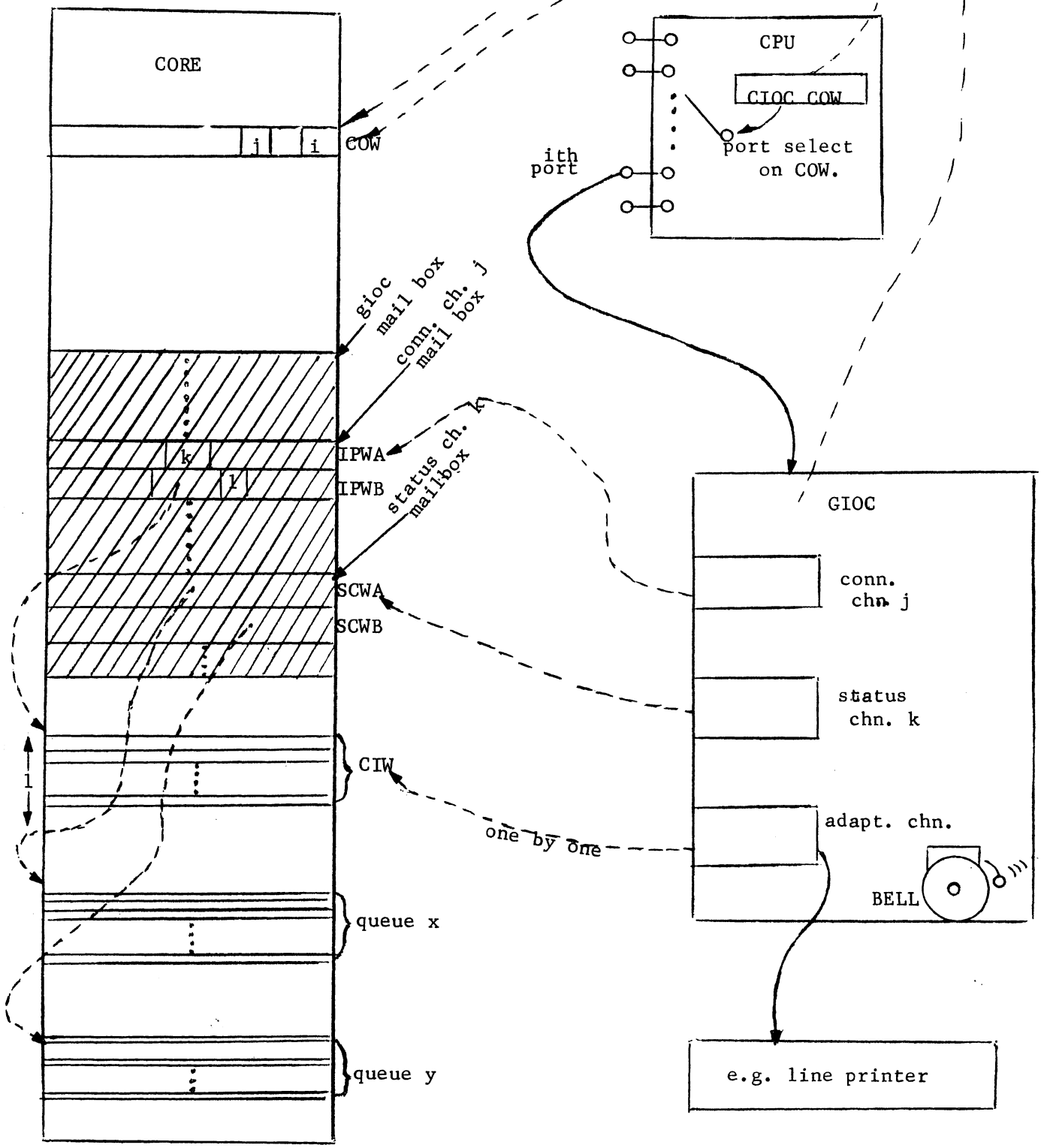

Fig. 4: GIOC Connect and Status Report

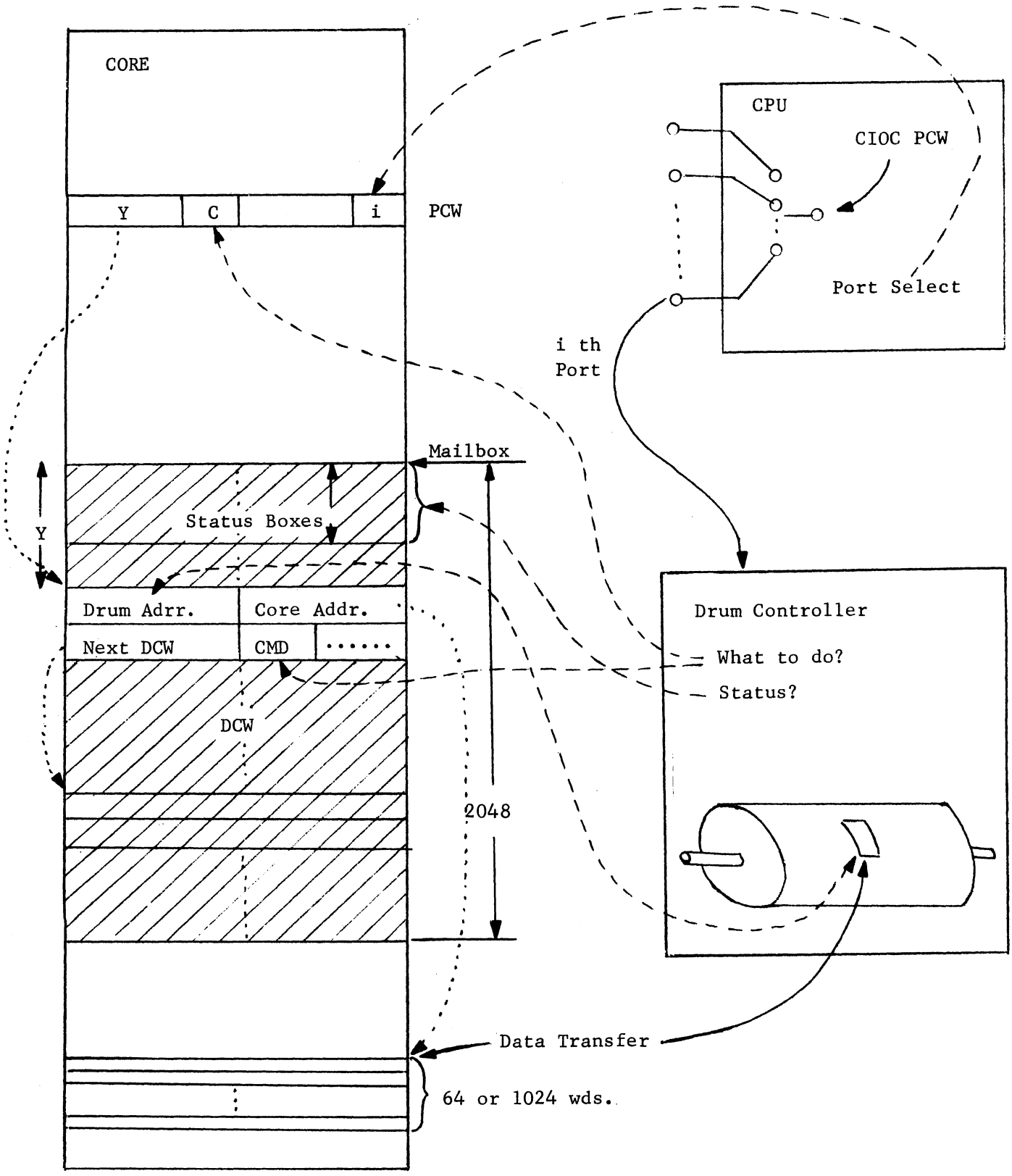

Fig. 5 Drum Controller Connect and Status Report

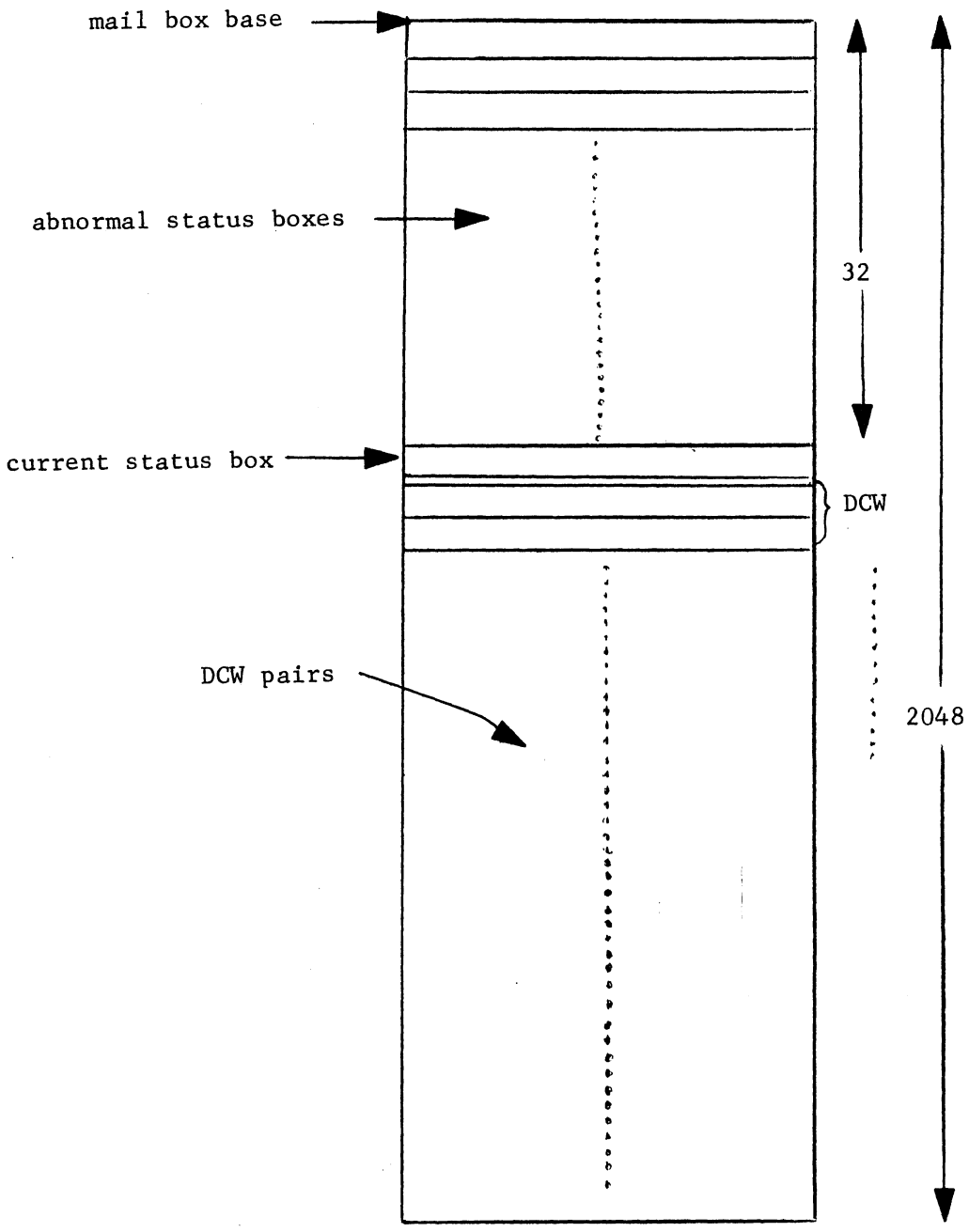

Fig. 6: Drum Controller Mail Box