1-1-1977

# A simulation model of the IBM customer information control system.

Donald S. Hoch

# A SIMULATION MODEL OF THE IBM CUSTOMER INFORMATION CONTROL SYSTEM

by
Donald S. Hoch

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Industrial Engineering

Lehigh University
1977

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

May 2, 1977
(date)

Professor in Charge

Chairman of Department

ACKNOWLEDGEMENTS

There are several people who have been of immense aid and assistance in the development of this thesis. My wife, Lucia, has always been ready with the necessary urging and has endured many inconveniences during the term of this paper. Tom Morrisette, a systems programmer at Pennsylvania Power and Light Company has provided me with much assistance and has patiently answered questions regarding CICS. Finally, my thesis advisor, Dr. Ben Wechsler, kept faith in me and continued to lead me along the path to completion. To the three persons named above and to any others who have provided me assistance, my gratitude.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

A SIMULATION MODEL OF THE IBM CUSTOMER INFORMATION CONTROL SYSTEM ... Donald S. Hoch

ABSTRACT

One of the measures of usefulness of an information system is its ability to process a request within a desired time frame. If, for some reason, the system is unable to respond within this time frame, then it loses all or part of its effectiveness.

This paper describes a simulation model for IBM's Customer Information Control System, an on-line computer system which processes inquiries and updates to a user data base. The inquiries and updates are initiated from telecommunications terminals and responses are directed back to these same terminals. If the time taken to respond to these transactions becomes too great, the system loses its effectiveness. This model can be used to discover those areas within CICS which act as bottlenecks given various input parameters.

Several simulation runs were made and their results are outlined within. One major problem discovered in these runs was in the routine which loads programs into core storage which are to be executed. Under certain circumstances this routine performs a considerable amount of extra work which is not required and which degrades the system to a great extent. A solution has been proposed for this problem.

# CHAPTER 1

# INTRODUCTION AND BACKGROUND

## 1.1 INTRODUCTION

To understand the reasoning behind and intent of this Master's thesis, it is necessary to delve into the background of on-line real-time computer systems, the reasons they came about and what they intended to accomplish. It would be best to start by defining what is meant by an on-line computer system. An on-line computer system is "one in which the input data enter the computer directly from the point of origination and/or output data are transmitted directly to where they are used. The intermediate stages of punching data onto cards or paper tape or of writing magnetic tape or off-line printing are largely avoided." [ 1]

On-line systems came about primarily because computers could not always provide information to a user within the time frame when it was most needed. This was true for a number of reasons:

1) Not every person or even company could afford one of these expensive machines, and hence, they might be forced to use a computer situated some distance away. Thus, there was the problem of getting the input to the computer and output returned from the computer within a reasonable period of time.

2) Once the data reached the computer, the problem still existed of scheduling and coordination of the necessary

events in order to get usable output.

An on-line system, in itself, solves part of this problem, which is the getting of the data to and the output back from the computer in a short period of time. This is accomplished by transmitting the data over transmission lines between a main computer and terminals, which themselves could be computers. Now, due to electronic speeds, time to get a job to the main computer is measured in seconds, rather than minutes, hours or even days. However, the problem of producing the output within a short period of time still exists.

It is here that the concept of 'real-time' enters. A real-time system is defined as "one which controls an environment by receiving data, processing them, and taking action or returning results sufficiently quickly to affect the functioning of the environment at that time." [ 1] It is the concept of sufficiently quick results that is of the essence here; that is, the provision of a system which provides a response time within which the user can function effectively. Within this concept, response time cannot be given a universal value since it is dependent upon the application in progress. A response time in seconds or less may be necessary to control an industrial production system, versus only a response time of minutes or hours for some commercial or managerial functions. However, in all cases the system must meet the time-dependent needs of the user to be considered real-time. It should be noted here that while some authors consider a real-time system, one which has a response time of seconds or fractions

of seconds, in this work we shall use the term only in its more general sense.

The data processing system has now expanded from the centralized computer and its assorted auxiliary equipment to include such things as terminals and telecommunication lines and their controllers. Also, what was once a basic operating system has now been expanded into an extremely complex set of software routines, hopefully capable of controlling the system equally well under varying conditions of stress and various requirements of individual requests to the system. Many, if not all, of the manual tasks which were present under the former 'batch' method of computing have now been replaced with automated controls as part of these software routines. Also, the telecommunications controller has taken on some new tasks due to the additional capabilities of the system. Some of these new functions include:

1) Polling of terminals to determine which have a request on the central processor.
2) Analyzing where input and output messages are to be routed.
3) Queuing up requests on various components of the system.
4) Translation between external transmission code and internal processing code.
5) Checking for transmission errors.

It is the Customer Information Control System from IBM upon which attention will be focused. This system was chosen because of its widespread existence in installations and also because the author intends on making practical use of the results of this system study. The above system is "a transaction-oriented, multiapplication data

base/data communication interface between a System/360 or System/370 operating system and user-written application programs. Applicable to most on-line systems, CICS provides many of the facilities necessary for standard terminal applications: message switching, inquiry, data collection, order entry, and conversational data entry." [ 2]

## 1.2 PERFORMANCE EVALUATION TECHNIQUES

As described in detail by Baafi [ 3] and mentioned by Shulman [ 4] , there are three performance evaluation techniques available for a computer system:

1) Monitoring (hardware or software)
2) Analytic evaluation
3) Simulation

Each particular method has its own advantages and disadvantages which must be weighed together with the system to be evaluated. For further information refer to table 1.1 for the major advantages and disadvantages of each method.

With respect to the system being studied, it can be seen that the only feasible technique is that of simulation. A hardware monitor is not available for CICS and is not worth any further consideration. It would be much too expensive to attempt to develop a hardware monitor to study the installation upon which this paper is based. Software monitors of the system are available, but they have two major drawbacks:

1) They only provide the user with an analysis of the system operation, and with no means of interpreting the analysis.

TABLE 1.1

Performance Evaluation Techniques Advantages/Disadvantages

| Evaluation Techniques | Advantages | Disadvantages |
|---|---|---|
| Hardware monitor | Has no effect on operation of system | Costly, not readily |
| Software monitor | Easy to write and change | Affects system operation |
| Analytic evaluation | Can be changed to model different situations | Difficult and sometimes impossible to construct |
| Simulation | Flexible for present and future operation | Costly to construct and run |

For example, they might indicate that the system performance degraded quite substantially at a particular point in time, but they give no indication as to the actual cause of the degradation.

2) Software monitoring does not provide a means of a priori measurement of statistics; that is, measurement before actual changes have been made to the system.

Analytic evaluation improves upon software monitoring in that it is possible to do a priori measurement, but an analytic evaluation of even some of the most simplistic systems is extremely time consuming. In the case of more involved systems, it has been impossible to develop an exact model of the system [ 5] . The complexity of CICS with its many possible interrelationships rules out this form of evaluation.

Simulation has been chosen because it is the best suited method for studying CICS. Refer to table 1.2 for the major advantages and disadvantages of simulation as prepared by Maisel and Gnugnoli [ 6] . The major reason why simulation is the best tool for performance evaluation for this system is that it can model the system comparatively easily. Another reason is that it has the extra and vastly important feature of flexibility. Not only can one consider many, if not all, states of a model in a properly prepared simulation, but one can easily adjust the model to test future system configurations and parameters.

## 1.3 OBJECTIVES

The main objective of this study will be to write and implement a simulation model of the IBM Customer Information Control System.

TABLE 1.2*

Summary of Advantages and Disadvantages of Computer Simulations

| Advantages | Disadvantages |
|---|---|
| Permits controlled experimentation with: | Very costly |
| (a) consideration of many factors | Uses scarce and expensive resources |
| (b) manipulation of many individual units | Requires fast, high capacity computers |
| (c) ability to consider alternative policies | Takes a long time to develop |
| (d) little or no disturbance of the actual system | May hide critical assumptions |
| Effective training tool | May require extensive field studies |
| Provides operational insight | |
| May dispel operational myths | |
| May make middle management more effective | |

* Refer to page 5 of Maisel and Gnugnoli 6 .

In order to do this, a comprehensive study will be done of the on-line environment of CICS as it is operating at Pennsylvania Power and Light Company. As a secondary objective the simulation model will then be used to determine the areas of the system which act as bottlenecks under normal and peak-load operating conditions. As a consequence of these bottlenecks, the response time of the system is often impaired. Once these bottlenecks have been found, solutions for them will be proposed. These solutions will then be tested by using the simulation model in order to ensure that no other potential bottleneck is created. Also, because of the flexibility and ease of modification inherent in simulation models, the model will be used to study the effect possible future revisions will have on the CICS system before the time and expense is incurred in making the revisions.

CHAPTER 2

# CICS ENVIRONMENT

## 2.1 GENERAL DESCRIPTION

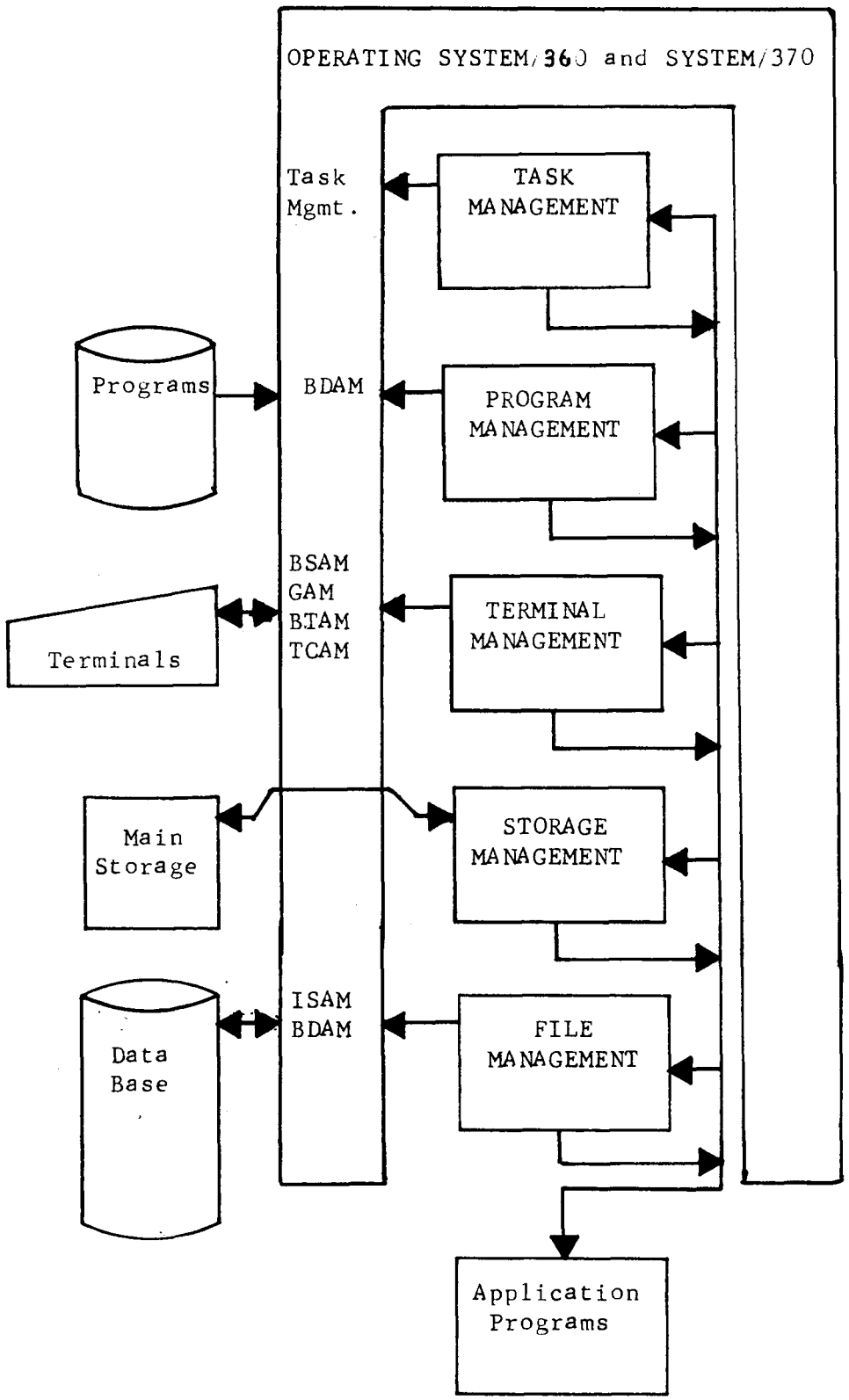
CICS consists of ten functional program modules which can be groomed by a user to meet his exact specifications. These modules interact with user-constructed tables to control the CICS environment and to process the user application program requests. Another main component of the CICS environment is the set of system input/output datasets which are used to support the real-time environment, and the user input/output facilities such as the terminals for interacting with CICS and the user data base. Even though CICS is a control system in its own right, it still must interact with and operate within the restrictions of the operating system on the System/360 or System/370. Refer to Figure 1 for a conceptual diagram of a CICS system.

## 2.2 FUNCTIONAL COMPONENTS

As mentioned above there are ten functional modules provided for a CICS system. They are:

1. Task Management
2. Storage Management
3. Program Management
4. Terminal Management
5. File Management
6. Transient Data Management
7. Temporary Storage Management
8. Program Interrupt Management
9. Time Management
10. Dump Management

CICS/OS SYSTEM FIGURE 1




* Refer to page 18 of IBM's CICS General Information Manual

In the discussion of these facilities it should be noted that in a CICS environment the words task and transaction are used synonymously.

### 2.2.1 TASK MANAGEMENT

Task Management provides the ability to process multiple transactions concurrently by use of a Task Control Program. This facility schedules and initiates processing of available tasks according to priorities assigned by the user and entered in one of the system control tables. When the task is complete, task management removes the task from the CICS environment. Also, the task program can dynamically change the priority of a task, and can delay the execution of a task by enqueueing it in order to synchronize the task with some other event in the CICS system. This event might be the notification of the completion of an input/output event or the request for a different task to be purged from the system. Task Management uses this enqueueing facility to control the number of active tasks processing within the system by not allowing any new tasks to be initiated once a user-supplied limit has been reached or if the amount of available main storage is insufficient to support those tasks already present.

### 2.2.2 STORAGE MANAGEMENT

Since CICS is a multitasking system, that is a program operating within a multiprogramming environment which is multiprogramming within itself, it is necessary for CICS to sub-allocate any resources which the operating system has

allocated to it. It is the function of the Storage Control Program of Storage Management to dynamically acquire and free main storage as requested by CICS system routines and user-written application programs. The main storage which is requested may be used for input/output areas, program load areas, system work areas or transaction work areas. Once the main storage area has been acquired, it may optionally be initialized to any desired bit configuration. For example, storage acquired as an output area might be initialized to all blanks before further processing.

A request for main storage may be issued by the user in one of two modes, either conditional or unconditional. If a conditional request is made and there is insufficient main storage to handle the request, Storage Control only returns control to the user with an indication that the request has not been satisfied. However, if the request was unconditional and there was insufficient storage, then Storage Control will take the following actions. It will:

1. suspend the requesting task until more core storage becomes available,
2. inhibit any new transactions from being initiated, and
3. release what is known as a 'Storage Cushion'.

A 'Storage Cushion' is an area of main storage which is held in reserve by CICS until a short-on-storage condition arises. At this time it is released and those transactions which are already in progress use it to satisfy their requests. The storage cushion is returned to a reserve status and new transactions may again be initiated whenever the short-on-storage condition has

been abated. This occurs whenever enough transactions have completed their processing and have been purged and the demands for main storage have decreased. If an additional request for core storage is issued when there is no more available storage in the cushion, then a stall condition may arise. If this is the case, then CICS can purge those tasks with low priority in order to allow the higher priority tasks to continue to process. The tasks which are purged are lost to the system and must be re-entered.

Another important function of Storage Control is to chain all acquired storage for a task together. This allows CICS to easily release any storage still owned by a transaction when that transaction ends, either normally or abnormally.

### 2.2.3 PROGRAM MANAGEMENT

Program Management is the area of CICS which supports the multiprogramming of transactions which is necessary in a real-time system. The Program Control program is responsible for dynamically loading, deleting, transferring control to and returning control from a program in the CICS environment. Program Management aids in the efficient use of main storage by allowing concurrent use of the same program 'copy' in main storage by multiple transactions. The only restriction imposed by the system is that the programs must be written in at least a quasi-reentrant manner. A fully reentrant program is one which does not alter any of its instructions or data during its execution, whereas a quasi-reentrant program is allowed to alter instructions or data, but it must restore anything

that has been altered to its original form before an exit is made from the program.

Program Management controls the programs in main storage by using a table known as the Processing Program Table. This table contains an entry for every program known to CICS. The entry contains the program's address in a direct access library, its address in main storage if it is currently resident and a use count indicating whether a program is currently active. Once loaded, a program remains in main storage until there is a short-on-storage indication. At that time any programs which are not currently in use are purged. When they are again required, they must be re-loaded into main storage.

### 2.2.4 TERMINAL MANAGEMENT

One of the necessary ingredients for a real-time on-line system is its terminal configuration and communications lines. Terminal Management is the area within CICS responsible for the control of this telecommunications network and which "provides for communication between terminals and user-written application programs through the Terminal Control Program." [ 2] The Terminal Control programs interacts with the Terminal Control Table when performing its duties in order to obtain information regarding the terminal device type, input/output access method to be used and line control data. Terminal Management also performs reads from and writes to the terminals, and converts the data, if necessary, to internal or terminal code.

### 2.2.5 FILE MANAGEMENT

As mentioned before, a prime component of CICS is the

user data base. The File Control Program, using the File Control Table, controls the input from and the output to the data base. File Management supports two types of IBM datasets, the Indexed Sequential Access Method and the Basic Direct Access Method. The Indexed Sequential Method, an indirect accessing scheme, constructs one or more indexes which refer to the position within the dataset where the desired physical record is located. Once the physical record has been located, it is directly read and the File Control program performs deblocking, if necessary. In contrast to the Indexed Sequential Method, the Basic Direct Access Method calculates the position of the physical record in the dataset by performing various functions on a user-supplied key. Through the interaction of these two access methods, File Management has the capability of presenting the records from a dataset to the user in either a random or sequential manner.

File Management also provides for the protection of the data base through a feature called 'exclusive control'. Exclusive Control prevents two or more transactions from concurrently attempting to update a logical record by enqueueing all transactions after the first which request a 'read for update' operation to the same logical record. Note that this does not imply that multiple transactions may not be updating the dataset concurrently, provided that each transaction is attempting to update a different logical record.

### 2.2.6 TRANSIENT DATA MANAGEMENT

Transient Data Management provides a means within CICS

for accumulating and transmitting data to terminals other than the one which initiated the task, to a dataset either within or outside of the CICS environment or to a program for subsequent processing. Those destinations which are within the CICS environment and which can only be accessed by CICS transactions are referred to as intrapartition destinations. Intrapartition destinations are queues of data which reside on a direct access device for eventual disposition to a CICS-related facility. Those destinations which are outside of the CICS environment are referred to as extra-partition destinations. These destinations may be datasets residing on either magnetic tape or direct access devices.

Transient data which has been sent to an intrapartition queue can be used to automatically initiate a transaction to handle the data. Whenever the number of records in the queue reaches a pre-defined level, the transaction is automatically initiated. This concept is known as a 'trigger level'. An example where this concept could be used would be in a process such as message switching. A transaction could perform a transient data write to an intrapartition queue which has a trigger level of one. This would automatically initiate a transaction which would read the data from the queue and send it to a specified terminal or group of terminals.

Extrapartition queues could be a dataset used to collect statistics or act as a transaction log for CICS and which would be examined at a later time. They could also be datasets used to collect or batch data being entered from remote terminals

and to be used for eventual offline processing.

### 2.2.7 TEMPORARY STORAGE MANAGEMENT

Liek Transient Data Management, Temporary Storage Management provides a facility within CICS for allocating and controlling working space for transactions which are being executed. However, unlike Transient Data Management, Temporary Storage Management is used when working storage (either main storage or direct access storage) is needed for use within the processing of a transaction. This allows the user to conserve main storage during the course of a transaction, a very important consideration in a system where the demands for storage space, at any one point in time, could far exceed that which is available. Also, this allows more transactions to be active concurrently, and increases the overall system throughput. The ability to multitask to a greater degree increases the overall system resource utilization, even though the time to process an individual transaction may be slightly increased. This has the effect of reducing the queue length of those transactions waiting to be initiated and of increasing the queues for many of the system resources. Also, from the viewpoint of the terminal operator, this generally has the effect of reducing the response time, a much sought-after attribute of an on-line real-time system.

### 2.2.8 PROGRAM INTERRUPT, TIME AND DUMP MANAGEMENT

The three remaining functional modules of CICS, Program Interrupt Management, Time Management and Dump Management,

provide important services to CICS, although they do not have the complexity of the previous seven which have already been discussed. In fact, all three functional areas are only optional features within CICS and are not necessary to have a functional on-line CICS environment.

Both System/360 and System/370 of IBM achieve their multiprogramming capabilities through a device known as an interrupt. An interrupt provides for the transfer of control of a computer system from a user's application program to the supervisor when certain exceptional conditions arise. There are five exceptional conditions which can trigger an interrupt in the System/360 or System/370 operating scheme. They are: input/output events, unusual program conditions, supervisor call or service requests, hardware errors and external conditions such as operator requests. The Program Interrupt Management facility of CICS intercepts and analyzes all interrupts caused by unusual program conditions within CICS. The normal action which the System/360 or System/370 operating system takes for an interrupt such as this is to abnormally terminate the program. However, this would mean that the entire CICS environment would be abnormally terminated. In its analysis of these program check interrupts, Program Interrupt Management determines which task was responsible and only abnormally terminates that task.

The Time Management function of CICS provides for many time-dependent functions to assist in the operation of CICS. It is used to determine when the transactions within the

system are at an impasse and to initiate corrective action as well as to detect and terminate a task which appears to be in a program loop. In addition, by using Time Management transactions can be made to wait for a specified period of time, can be notified after a specified time interval has elapsed or even be automatically initiated after a specified time interval or at a particular point in time.

Dump Management is used to write out to output datasets images of main storage, such as program working storage, input/output areas or system tables. Normally, this is most important when used in a testing environment, but it is also useful in error analysis of a task which has been abnormally terminated.

## 2.3 SYSTEM TABLES

As mentioned before, CICS uses a number of user-constructed tables to control its environment. These tables define for the CICS system all of those elements which in total comprise the CICS environment. The main tables and their components are described below. The Program Control Table contains one entry for each valid transaction code available to the system. Coded into the table is:

1. the priority and security code required by each transaction, and
2. information necessary for the processing of each transaction, such as the first program to be called by the transaction.

The Processing Program Table contains one entry for each application program available to the system. The information

contained in the entry describes the program source language and whether the system is resident or non-resident in main storage.

The Terminal Control Table is used to define the user's terminal environment. There is one entry for every terminal, communications line or control unit available to the system. Each entry contains descriptive information about the device or line and also various device dependent characteristics.

The File Control Table describes the user's data base which is available to CICS, with one entry needed for each dataset to be accessed. Each entry describes in detail all characteristics of the dataset.

The Destination Control Table describes the environment which is accessible by Transient Data Management. It contains one entry for each intrapartition or extrapartition destination. The entries are used to describe the characteristics of the destinations.

While CICS contains other optional tables and facilities, a description thereof is not necessary to an understanding of system operation and will not be discussed. An interested reader is referred to references [ 2, 8, 9] .

CHAPTER 3

# SIMULATION MODEL OF CICS

## 3.1 INTRODUCTION

The writing of the simulation model for CICS entailed two major and distinct steps:

1. The standard FORTRAN-based GASP2 routines as used at Lehigh University were "translated" into PL/I, a programming language with which the author is much more familiar. In addition to the translation, several modifications were made to the routines which were felt to be necessary for successful implementation of the CICS model.
2. The user subprograms were written which represented the CICS system being modeled.

In preparing this simulation model it was necessary to make some underlying assumptions in order to limit the scope of the problem to a model of CICS. Also, some assumptions were necessary to obviate the need of becoming involved in trivialities. The primary assumption was that only those problems which were directly relatable to the CICS environment were studied. Secondary assumptions were that all resources were operating with no mechanical problems and that all code in all programs was efficient,

"user" code. For a more detailed discussion of these assumptions, the reader is referred to Appendix B.

Immediately following will be an overview of the user subprograms which are used to model CICS. A discussion of the GASP2 to PL/I translation can be found in Appendix C while a more detailed discussion of the user subprograms can be found in Appendix D.

## 3.2 CICS MODEL

The CICS model which has been developed consists of 30 user-written subroutines in addition to the 21 GASP2 system subroutines, the user-written event selection subroutine and the user-written OTPUT routine. There are three input files to the model with the following functions:

1. The standard GASP2 input cards containing run control data, parameter data, file data and initial events.
2. The file containing the initialization data for the Processing Program, Program Control and File Control tables.
3. The file containing variable parameter data for the system, such as the number of core storage blocks available to be allocated, the mean time between transaction arrivals, various parametric distributions and so forth.

These files will be referred to throughout this

discussion as the GASP2 Event file, Initialization file and Parameter file, respectively. There are also seven internal "files" which are stored in the GASP2 filing array QSET. These files are discussed more thoroughly in Appendix A.

Perhaps the best way to explain this complex set of procedures is to discuss the routines as they might be encountered during a normal run of the model, while referring to a logic flowchart of each routine.

A main program (Figure 2) essentially just initializes variables before it exits to GASP2. The first initial event to be entered into the GASP2 Event file causes routine SYSINIT (Figure 3) to be executed. This routine models the start-up of CICS. The second and final initial event to be entered into the GASP2 Event file calls routine TC_NEXT (Figure 4). TC_NEXT is used to schedule the time of the next terminal requesting transaction processing. The routine also is used to determine the transaction type. An event for this transaction is entered into the GASP2 Event file. When this entry is removed from the file, it calls routine TC_GET (Figure 5). This routine simulates the actions necessary to perform a read from the terminal.

After completing these actions, it inserts an event into the GASP2 Event file with an event time equal to the current time. This event calls routine KC_A (Figure 6), the Task Control Attach routine. This routine simulates all the necessary actions required to initiate a new task in CICS.

After completing all its actions, routine KC_A exits to Routine DSPTCHR (Figure 7). This routine is the hub of all CICS processing because it determines what task is to have use of the CPU in order to execute. After a task has been Using the CPU for a period of time, various conditions may arise which cause it to be suspended or placed into a wait state. A task is suspended in routine KC_S (Figure 8) because some condition prohibits the task from proceeding beyond its current status. A task is placed into a wait state in routine KC_W (Figure 9) because some specified event must be completed before it can continue. When that event has been consummated, routine KC_R (Figure 10) is used to resume the active execution of the task. When the task has been completed, either normally or abnormally, routine KC_T (Figure 11) simulates the actions necessary to purge the task from the system. Two other functions are modeled by the system at the task level. Routine KC_C (Figure 12) is used to dynamically change the dispatching priority of any task in the system, while routine KC_RS (Figure 13) is used to model the testing of various resources to determine their state.

Several routines are used to model CICS actions at the program level. They should not be confused with those routines at the task level, because it may take several programs to perform all the actions required by one task. Routine PC_F (Figure 14) models the first necessary action,

the loading of a program into main storage. A program in CICS can be branched to in two ways. It can be linked to or have control transferred to it. Routine PC_L (Figure 15) models the actions of linking from one program to another, where linkage is set up in order to return to the calling program. In order to simulate transfer of control, routine PC_R (Figure 16) models the release of the currently-allocated program in order to start execution of the next program. If, due to a short-on-storage condition, it is necessary to obtain more core storage, routine PC_D (Figure 17) simulates the deletion of unused programs from core storage residency. The last remaining function of program control is to handle the abnormal termination of user-written programs. Routine PCABEND (Figure 18) simulates those actions.

Storage Control routines are used by both user-written and CICS system routines to perform and monitor the allocation and deallocation of core storage. Routines SC_O (Figure 19) and SC_OS (Figure 20) model the allocation of core storage, while routines SC_R (Figure 21) and SC_FS (Figure 22) simulate the deallocation of core storage. Routine SC_F (Figure 23) models the monitoring of core storage usage, and attempts to restart any task which has been suspended due to a short-on-storage condition.

Temporary Storage Control in CICS provides a means of

controlling task working space which is to be used for an extended period of time. Routine TS_P (Figure 24) simulates the allocation and writing of a block of data to a Temporary Storage file. Routine TS_GR (Figure 25) simulates reading a block of data from the Temporary Storage file, and/or deallocating that block from task ownership.

The routines of File Control perform all operations necessary to communicate with the user data base. Eight major areas are simulated by the model. Routine FC_OCL (Figure 26) simulates the actions of creating a linkage between a task and a user file and of removing the linkage. Routine FC_S (Figure 27) models the creation of several file work areas and control areas which make it possible for a task to retrieve records in a logically sequential order from a direct-access type file. Routine FC_GN (Figure 28) models the retrieval of a record from a sequential order as defined by routine FC_S. Routine FC_RES (Figure 29) is used to model the re-initialization of the work and control areas used in sequential record accessing when a different sequential string of records is desired. The two routines which simulate the input and output of records in a direct fasion are FC_GET (Figure 30) and FC_PUT (Figure 31). Routine FC_GA (Figure 32) models the actions necessary to initialize and create a record to be written to the user data base. Finally, routine FC_RL_E (Figure 33) models the release of work and control areas obtained for all the above file operations.

The remainder of the routines can be categorized as either service routines or miscellaneous routines. DMPCNTL (Figure 34) models the actions necessary to "dump" all or portions of a task's storage to an output device when a program in the task has encountered some condition which forces it to terminate abnormally. Routine OS_WAIT (Figure 35) models the condition of placing CICS in a wait state and branching to the computer's supervisory program when there is no active work which can be performed within CICS. Two routines of the model are used to simulate the start and completion of input/output events. Routine READWRT (Figure 36) models the initiation of the input/output event, while routine OS_POST (Figure 37) models the actions taken upon completion of an input/output event, as well as other conditions which cause wait states. Finally, routine END_SIM (Figure 38) is used to close out all time-generated statistics being collected in the model when the model has operated for its intended period of time. It also sets indicators to notify the GASP2 control routines to end the simulation and prepare the summary reports.

## 3.3 FREQUENCY DISTRIBUTIONS IN THE MODEL

The basic piece of information around which CICS functions is the task. Therefore, a model of CICS must have a way of representing this information. The two characteristic pieces of data which the model uses to represent tasks are the time of arrival and the task type.

Normally, in modeling a system such as CICS, statistics and observations from the actual system are used to provide insight into the distributions of arrival time and task type. However, in the CICS system being observed, there are no detail statistics at the task level which are available. The only statistics available are gross summaries which are printed out at the end of the operating day, indicating such things as total number of tasks processed by the system and a breakdown of the total by task ID. This information is not sufficient to hypothesize a distribution of task type because the work that a task does is variable in nature. That is, the parameters supplied to a particular task may, in one instance, need one input/output event to determine the answer and another set of parameters for the same task may require 15 input/output events to determine the answer. For this reason it was decided that there was no present method of developing a distribution of task types for the system being observed, and each task type was given an equal chance of being the one selected by using a uniform distribution.

A similar situation exists for the time of arrival of tasks. There is even less available information from which to develop this distribution. For this reason it was decided to use the Poisson distribution to describe the arrival times of tasks. The Poisson distribution is used to describe the probability of N events occurring per time unit, where N is

this case would be the average number of tasks presented to the system per second. The Poisson distribution gives equal probability of a task arriving at one point in time as in another, and as such implies that there will be no peaks or valleys in demand for the system. This may not be the case, but no data is available to prove otherwise. It has been shown that if the arrival times of events are Poisson distributed, then the time between arrivals is exponentially distributed [ 15] . This exponential distribution is used to determine the point in time when the next task will arrive.

## 3.4 VALIDATION OF THE MODEL

According to Webster's New Collegiate Dictionary, validity is "the state of having a conclusion correctly derived from given premises". In modeling, be it mathematical or simulation modeling, the output of a model is only as credible as the model is valid. This implies that before the task is undertaken to construct a model, it is mandatory to know what is expected as the output or results of the model. Said in another way it is necessary to define objectives prior to constructing the model upon which the model can be predicated. The stated objective of this simulation model was to use it to gain insights as to the possible areas and causes of backlogs in IBM's Customer Information Control System.

Given the stated objective, it is necessary to determine whether the simulation model output satisfies the objective. As with any computer program, attention was given from the onset of computer programming to precisely representing the system in a computer language. Until all programming language errors are eliminated, it is impossible to proceed further with the validation. After all coding errors had been found and corrected, it was possible to make simulation runs to determine the logical consistency of the model; that is, whether all subprograms in the model accurately represent the corresponding subsystem in CICS. The running of the model was done under the control of the PL/I Checkout Compiler, as was the running of the GASP2 routines which were translated from FORTRAN into PL/I. The great flexibility of the compiler increased the ease and shortened the time span in finding logical inconsistencies within the model. After all this was done, it was now possible to determine whether the model met its objective.

Several distinct options have been expressed on the subject of what constitutes a validation of a model. In reference[ 6 ] by Maisel and Gnugnoli, three separate sets of checks are suggested:

1. Use parameters in place of constants to facilitate modification of the model to meet changes in the system being modeled.
2. Get expert opinion as to the closeness of representation of the model to the real system.

3. Compare model results to known standards by statistical measures.

It is further suggested that a combination of all three sets of checks might produce the best validation of the model. However, other authors do not share the same confidence in these checks. The use of parameters in model equations is nothing more than good computer programming practice and will in no way guarantee a better end result. Also, as stated by P. H. Seamon in reference [17], "estimators obtained from the model cannot be taken as predictors of absolute performance' if necessary input variables or parameters are not available at the time the system is modeled. This would be true for a simulation model having many independent and dependent variables as in this study's model. Also, due to the complexity of the system being modeled, a set of standards may not be available. Any estimators from this CICS model could at best be labeled suspect if they were to be used as predictors of validity due to the inavailability of known details about the CICS system.

Another author, Jay W. Forrester, in reference [18], along with P. H. Seamon in reference [17], takes a much different approach to the question of validity. They do not think it is necessary to validate a model by statistical means to known standards. Forrester is even much more outspoken about quantitative validation, in that he believes

it to frequently be a matter of delusive exactitude, a matter of attempted validation which should only be done under certain conditions. He believes that quantitative measurement should only be performed if the work and cost involved in collecting the standard data is not significant. If any shortcuts are taken to minimize the time and money involved in collecting the data, then the data would probably be suspect and no true validation would be performed.

It is the concept of these authors that the validity of a model should be judged finally on the model's ability to accomplish its stated objective. It is Seamon's feeling that a model need not be able to produce absolute results, but be able to give the user a feeling of relative results when changing the model from one state to another. Forrester sets forth several criteria which he feels necessary in the validation of a model. His first criterion for validation is that the model show no obvious inconsistency with observed actual data. Although this sounds trivial, Forrester states that most models which he has examined have not kept this criterion in mind. His second criterion used in model development is to initially attempt to make the model plausible with its results, not 100% accurate. This approach emphasizes the main intent of developing a model, to learn as much as possible about the system being modeled. A model need not be developed to the point of accurately modeling a

system, only to the point where a plausible relationship exists between the model and the system being modeled so that the model can be put to use. His last criterion stresses that in lieu of using quantitative measurement techniques, many models should be validated by gleaning knowledge and intuitive concepts from the model's author and a team of experts in the field. It is his hypothesis that to validate an area of study which cannot be expressed numerically requires the validation to take on a non-numerical approach. He feels that this collection of knowledge being concentrated on the model will, in the end, justify it as being representative of the system being simulated, and may even do it at a faster pace than quantitative measuring would by itself.

For several reasons the model of CICS developed in this study was validated using the concepts of Forrester and Seamon. The statistics available from the CICS system being observed were only available at a very high level. This meant that a large number of the figures needed to run the model would be pure estimates or educated guesses, and the output statistics would be meaningless as absolute numbers. Also, one other problem area which would have inhibited quantitative validation was the fact that the CICS system being studied was being run in a multiprogramming environment which would have introduced an unknown amount of noise

into the statistics. Finally, the finished model was quite complex and extensive, and the facilities were not available to validate the model in any other manner. For these reasons it was deemed necessary to follow the criteria of Forrester in validating the model. Several modeling runs were made which were examined for plausibility and consistency with what would be expected. Also, a systems programmer at the installation being examined was referred to for his opinion and counseling on the model and its output. The systems programmer was responsible for maintaining and enhancing the CICS network at the installation for many years, and could easily be qualified as an expert in the field. Appendix F is a letter of testimony written by this system programmer stating his opinion on the validity of this model.

CHAPTER 4

# ANALYSIS OF PROBLEMS

After reaching the point in the development of the simulation model that it accurately represented the real world, it was necessary to use the model to gain insights into IBM's Customer Information Control System. This was the final step in determining whether or not the model satisfied its design objective.

An initial simulation run was made with an estimate of various system parameters. The time increment used in the model was milliseconds, one one-thousandths of a second, and the model was executed for 60,000 time intervals. Data was accumulated for three GASP2 COLCT-type statistics, as well as seven GASP2 TMST-type statistics. The three COLCT-type statistics are:

1. Total time in the system for a task,
2. Wait time in the system for a task, and
3. Core storage usage.

The seven TMST-type statistics are:

1. Percent of time that the program loader is active,
2. Number of active tasks in the system,
3. Percent of time that no task may be attached for any reason,
4. Percent of time that no task may be attached because the system was at MAX TASK,
5. Percent of time that no task may be attached because

the storage cushion is allocated,

6. Number of tasks queued, and

7. Percent of time that CICS is idle because there are no dispatchable tasks.

Table 4.1 is used to give a synopsis of these statistics. For the interested reader, all computer listings, including the program compile and all simulation runs referred to in this paper, are available at the Lehigh University Industrial Engineering Department library.

As mentioned previously, one of the possible areas of concern which could be studied was core storage usage and its effect on response time and throughput. From the initial simulation run it can be seen that approximately 11 percent of the time the system was prohibiting new tasks from being attached because of a short-on-storage condition. As an attempt at lowering this percent and achieving better response and more throughput, the primary core storage allocation was increased by 25 percent and a second simulation run was made with all other parameters remaining unchanged. The statistics for this run (run 4.2) are displayed in Table 4.2.

At first glance comparison of the two sets of statistics appears to reveal several incongruous facts. For example, even though the amount of core storage was significantly increased, the percent of time that the system was in a short-on-storage condition was relatively the same (11.0% versus 10.7%). Also, even though the throughput improved (203 completed tasks versus

228), the average time to process a task increased by almost 40 percent (1,240 milliseconds versus 1,725 milliseconds).

As an attempt at explaining these apparent puzzles, several explanations can be proposed. On the average there are about 50 percent more tasks in the system in run two than in run one at any point in time (4.2 tasks versus 6.1). This means that there will be more tasks vying for all system resources, not just core storage. This is evidenced by the fact that there are slightly more tasks enqueued and suspended in run two than in run one (1.6 tasks versus 1.8). This is also proven out by comparing a statistic calculated by taking the difference between the average system time and the average wait time. This statistic represents the amount of time spent executing a task, on the average, disregarding any time spent waiting or being suspended. The execution times in run one and run two compare favorably (149 milliseconds versus 153 milliseconds). This implies that the increase in the response time was strictly due to an increase in the time spent enqueued.

Also, it appears that the increase in system throughput is entirely due to the added core storage. The increase in the number of active tasks must be due entirely to the added core storage, since that was the only parameter changed. This increase also resulted in the system utilization percent improving. The system utilization percent is calculated as follows:

1.0 - system idle time.

for run one the figure is approximately 93 percent and for run two 97.9 percent. Finally, the program loader was active about 2.3 percent less in run two than run one (46.8% versus 44.4%). This is significant because this indicates that less programs had to be loaded in run two than run one because they were already resident in core storage when needed. The combination of these factors can explain the greater throughput in the second run.

In order to further validate these explanations of the changes between run 4.2 and run 4.1, two additional runs were made. The first run, summarized in Table 4.5, represents an addition of the core storage available by 100% over run 4.1, and the second run, summarized in Table 4.6, represents a reduction of the core storage available by 50%. These runs entirely support the explanations proposed in the preceding paragraphs. The average number of tasks in the system in run 4.5 and the average time to process a task are greatly increased. Also, the average number of queued tasks has increased, which in conjunction with the increased average response time, indicates the increased vying for other system resources. The average execution time (160 milliseconds) is still consistent with runs 4.1 and 4.2, as would be expected. However, the throughput has not increased because the system was at MAX TASK condition for 7% of the time. As expected, the percent of time in which the program loader was active is again reduced, due to the additional core storage available and also to the fact that the increased number

of active tasks will inhibit programs from being deleted if not being used.

Conversely, run 4.6 shows a sharply decreased throughput and a much improved response time. These statistics are keeping in line with the above discussion. Also, the percent of time that the program loader is active is again relatively high. However, the most revealing statistics are the percent of time short on storage (33.9%) and the percent of time when CICS is idle (22.3%). These two statistics explain the reduced throughput to a great extent.

However, one of the prime considerations of an on-line system of this type is to control and minimize the response time. If core storage was the only or even primary bottleneck within the system, then the 25 percent increase in core storage from run one to run two should logically have improved the response time. The opposite results imply that there are other factors affecting the response time more so than the amount of core storage available to the system. One area which certainly warrants further investigation is the relationship of a tasks total time in the system to its total wait time. In run 4.1 the percentage of time spent waiting was bout 88 percent while in run 4.2 it was about 91 percent. If this percentage could be reduced, the average response time would improve.

To determine how to reduce this percent, it is necessary to know exactly what factors make it up. The main reasons for a

task to wait in the model are:

1. Unavailability of core storage,
2. Waiting for use of the program loader,
3. Waiting for the completion of an input/output event,
4. Waiting for use of the Temporary Storage facilities.

As mentioned previously, the percent of time which the program loader was busy was relatively high (46.8% versus 44.4%) and since the period of time necessary to load a program is relatively lengthy, it is quite probable that a significant proportion of a task's waiting time is attributable to the program loader operation.

One way of improving the operation would be to decrease the number of program loads in a period of time by increasing the number of programs made permanently resident. A second way would be to reorganize the program libraries to give the optimum configuration for loading. A third way would be to optimize the program loader itself. After reviewing the situation with a systems programmer at Pennsylvania Power and Light who is familiar with the operations of CICS, a combination of the second and third methods was tried.

User programs for CICS can be written in either IBM's Assembler language or one of two high-level languages, COBOL or PL/I. Programs written in the high-level languages are stored in load libraries in executable form and have up to five control records preceding the first record of text. Under many

circumstances these control records are not needed and only act as overhead. It was this area which was attacked.

Additional runs were made with revisions to the program loader routine to encompass the above-mentioned change. Run 4.3 used the same parameters as run 4.1 and run 4.4 the same as run 4.2. Their results are outlined in Tables 4.3 and 4.4, respectively. By comparing runs 4.1 and 4.3, it can be seen that there is an improvement in some areas, but not the total task time, wait time or time with a short-on-storage condition. By comparing runs 4.2 and 4.4, all areas have improved. The total system time, wait time, percent of time at a NO ATTACH condition and total throughput are all at their best values.

Again, two corroborating runs were made similar to runs 4.5 and 4.6, only using the revised program loader routine. These runs, 4.7 and 4.8 again substantiate the original conclusions. Again, it should be noted that by merely increasing the core storage, as in run 4.7, one cannot continue to improve upon all conditions. Eventually, as has happened, a bottleneck will develop in some other area of the system, and the wait time will increase. At some point in time, all practical and relatively inexpensive improvements will have been made to the CICS environment and only such changes as upgrading the CPU or additional channels will improve performance.

This concept is easily visible in two final simulation runs, runs 4.9 and 4.10. Run 4.9 used the same parameters as run 4.4

except that the maximum number of tasks allowable within the system was increased from 20 to 25. This not only had the desired effect of reducing the percent of time at NO ATTACH due to a MAX TASK condition, but it also resulted in an increase in the response time and a decrease in the throughput, two nondesirable results. As was explained above, some other bottleneck has developed and affected the system in a negative manner. Run 4.10 parallels run 4.7 except for the increase in maximum allowed number of tasks from 20 to 25. This run likewise shows the development of a different bottleneck.

TABLE 4.1

COLCT-TYPE STATISTICS

| | MEAN | STD.DEV. | MIN | MAX | OBS |
|---|---|---|---|---|---|
| Total Time | 1240.3 | 757.2 | 131.9 | 4551.6 | 203 |
| Wait Time | 1091.2 | 678.6 | 85.3 | 4019.7 | 203 |
| Core Storage Usage | 18.9 | 5.6 | 5.4 | 36.5 | 4818 |

TMST-TYPE STATISTICS

| | MEAN | STD.DEV. | MIN | MAX | OBS |
|---|---|---|---|---|---|
| Percent Program Loader Active | 46.8% | 49.9% | 0.0% | 100.0% | 60000 |
| Active Tasks | 4.2 | 3.7 | 0.0 | 12.0 | 60000 |
| Percent of Time at No Attach | 11.0% | 31.3% | 0.0% | 100.0% | 60000 |
| Percent of Time at Max Task | 0.0% | 0.0% | 0.0% | 0.0% | 60000 |
| Percent of Time at Short-on-Storage | 11.0% | 31.3% | 0.0% | 100.0% | 60000 |
| Queued Tasks | 1.6 | 1.9 | 0.0 | 9.0 | 60000 |
| Percent of Time at CICS Idle | 6.9% | 25.4% | 0.0% | 100.0% | 60000 |

TABLE 4.2

COLCT-TYPE STATISTICS

| | MEAN | STD.DEV. | MIN | MAX | OBS |
|---|---|---|---|---|---|
| Total Time | 1725.8 | 1226.0 | 113.4 | 6426.9 | 228 |
| Wait Time | 1572.0 | 1145.8 | 66.6 | 5900.9 | 228 |
| Core Storage Usage | 22.7 | 7.1 | 7.0 | 48.1 | 5313 |

TMST-TYPE STATISTICS

| | MEAN | STD.DEV. | MIN | MAX | OBS |
|---|---|---|---|---|---|
| Percent Program Loader Active | 44.4% | 49.7% | 0.0% | 100.0% | 60000 |
| Active Tasks | 6.1 | 5.4 | 1.0 | 19.0 | 60000 |
| Percent of Time at No Attach | 10.7% | 30.9% | 0.0% | 100.0% | 60000 |
| Percent of Time at Max Task | 0.0% | 0.0% | 0.0% | 0.0% | 60000 |
| Percent of Time at Short-on-Storage | 10.7% | 30.9% | 0.0% | 100.0% | 60000 |
| Queued Tasks | 1.8 | 2.1 | 0.0 | 10.0 | 60000 |
| Percent of Time at CICS Idle | 2.5% | 15.5% | 0.0% | 100.0% | 60000 |

TABLE 4.3

COLCT-TYPE STATISTICS

| | MEAN | STD.DEV. | MIN | MAX | OBS |
|---|---|---|---|---|---|
| Total Time | 1252.0 | 954.1 | 1e6.1 | 347..3 | 217 |
| Wait Time | 1094.6 | 883.8 | 111.9 | 5073.2 | 217 |
| Core Storage Usage | 12.2 | 6.4 | 6.0 | 36.9 | 5708 |

TMST-TYPE STATISTICS

| | MEAN | STD.DEV. | MIN | MAX | OBS |
|---|---|---|---|---|---|
| Percent Program Loader Active | 20.0% | 40.0% | 0.0% | 100.0% | 60000 |
| Active Tasks | 4.4 | 4.6 | 0.0 | 20.0 | 60000 |
| Percent of Time at No Attach | 10.4% | 30.5% | 0.0% | 100.0% | 60000 |
| Percent of Time at Max Task | 0.9% | 9.3% | 0.0% | 100.0% | 60000 |
| Percent of Time at Short-on-Storage | 9.5% | 29.3% | 0.0% | 100.0% | 60000 |
| Queued Tasks | 1.0 | 1.1 | 0.0 | 7.0 | 60000 |
| Percent of Time at CICS Idle | 3.0% | 17.0% | 0.0% | 100.0% | 60000 |

TABLE 4.4

COLCT-TYPE STATISTICS

| | MEAN | STD.DEV. | MIN | MAX | OBS |
|---|---|---|---|---|---|
| Total Time | 918.0 | 701.0 | 163.0 | 4104.7 | 236 |
| Wait Time | 776.6 | 659.5 | 112.1 | 3803.9 | 236 |
| Core Storage Usage | 15.8 | 6.1 | 6.7 | 41.2 | 5724 |

TMST-TYPE STATISTICS

| | MEAN | STD.DEV. | MIN | MAX | OBS |
|---|---|---|---|---|---|
| Percent Program Loader Active | 19.9% | 39.9% | 0.0% | 100.0% | 60000 |
| Active Tasks | 3.1 | 3.9 | 0.0 | 20.0 | 60000 |
| Percent of Time at No Attach | 1.0% | 10.0% | 0.0% | 100.0% | 60000 |
| Percent of Time at Max Task | 0.1% | 3.7% | 0.0% | 100.0% | 60000 |
| Percent of Time at Short-on-Storage | 1.0% | 10.0% | 0.0% | 100.0% | 60000 |
| Queued Tasks | 0.9 | 1.1 | 0.0 | 6.0 | 60000 |
| Percent of Time at CICS Idle | 3.7% | 18.8% | 0.0% | 100.0% | 60000 |

TABLE 4.5

COLCT-TYPE STATISTICS

| | MEAN | STD.DEV. | MIN | MAX | OBS |
|---|---|---|---|---|---|
| Total Time | 2853.0 | 1968.2 | 420.5 | 14420.0 | 224 |
| Wait Time | 2693.1 | 1864.8 | 373.3 | 13656.2 | 224 |
| Core Storage Usage | 33.9 | 6.7 | 16.0 | 53.8 | 5695 |

TMST-TYPE STATISTICS

| | MEAN | STD.DEV. | MIN | MAX | OBS |
|---|---|---|---|---|---|
| Percent Program Loader Active | 38.2% | 48.6% | 0.0% | 100.0% | 60000 |
| Active Tasks | 10.5 | 7.8 | 7 | 20 | 60000 |
| Percent of Time at No Attach | 7.0% | 25.4% | 0.0% | 100.0% | 60000 |
| Percent of Time at Max Task | 7.0% | 25.4% | 0.0% | 100.0% | 60000 |
| Percent of Time at Short-on-Storage | 0.0% | 0.0% | 0.0% | 0.0% | 60000 |
| Queued Tasks | 2.0 | 2.1 | 0.0 | 10.0 | 60000 |
| Percent of Time at CICS Idle | 0.0% | 0.0% | 0.0% | 0.0% | 60000 |

TABLE 4.6

COLCT-TYPE STATISTICS

| | MEAN | STD.DEV. | MIN | MAX | OBS |
|---|---|---|---|---|---|
| Total Time | 1073.6 | 735.8 | 125.2 | 4144.1 | 132 |
| Wait Time | 916.1 | 655.6 | 71.9 | 3591.3 | 132 |
| Core Storage Usage | 13.4 | 4.7 | 5.0 | 28.9 | 3265 |

TMST-TYPE STATISTICS

| | MEAN | STD.DEV. | MIN | MAX | OBS |
|---|---|---|---|---|---|
| Percent Program Loader Active | 45.4% | 49.8% | 0.0% | 100.0% | 60000 |
| Active Tasks | 2.4 | 2.2 | 0.0 | 8.0 | 60000 |
| Percent of Time at No Attach | 33.9% | 47.3% | 0.0% | 100.0% | 60000 |
| Percent of Time at Max Task | 0.0% | 0.0% | 0.0% | 0.0% | 60000 |
| Percent of Time at Short-on-Storage | 33.9% | 47.3% | 0.0% | 100.0% | 60000 |
| Queued Tasks | 1.4 | 1.6 | 0.0 | 7.0 | 60000 |
| Percent of Time at CICS Idle | 22.3% | 41.6% | 0.0% | 100.0% | 60000 |

TABLE 4.7

COLCT-TYPE STATISTICS

| | MEAN | STD.DEV. | MIN | MAX | OBS |
|---|---|---|---|---|---|
| Total Time | 1800.1 | 1491.3 | 226.3 | 9234.1 | 246 |
| Wait Time | 1647.7 | 1428.0 | 175.1 | 8695.8 | 246 |
| Core Storage Usage | 22.4 | 8.5 | 7.6 | 59.9 | 6123 |

TMST-TYPE STATISTICS

| | MEAN | STD.DEV. | MIN | MAX | OBS |
|---|---|---|---|---|---|
| Percent Program Loader Active | 19.1% | 39.3% | 0.0% | 100.0% | 60000 |
| Active Tasks | 6.7 | 6.3 | 2.0 | 20.0 | 60000 |
| Percent of Time at No Attach | 3.4% | 18.1% | 0.0% | 100.0% | 60000 |
| Percent of Time at Max Task | 2.4% | 15.4% | 0.0% | 100.0% | 60000 |
| Percent of Time at Short-on-Storage | 1.1% | 10.4% | 0.0% | 100.0% | 60000 |
| Queued Tasks | 1.2 | 1.3 | 0.0 | 9.0 | 60000 |
| Percent of Time at CICS Idle | 0.2% | 4.3% | 0.0% | 100.0% | 60000 |

TABLE 4.8

COLCT-TYPE STATISTICS

| | MEAN | STD.DEV. | MIN | MAX | OBS |
|---|---|---|---|---|---|
| Total Time | 673.3 | 416.3 | 141.1 | 2499.4 | 172 |
| Wait Time | 520.4 | 358.6 | 81.0 | 2338.9 | 172 |
| Core Storage Usage | 12.1 | 4.1 | 5.2 | 27.1 | 4567 |

TMST-TYPE STATISTICS

| | MEAN | STD.DEV. | MIN | MAX | OBS |
|---|---|---|---|---|---|
| Percent Program Loader Active | 21.6% | 41.1% | 0.0% | 100.0% | 60000 |
| Active Tasks | 2.1 | 2.1 | 0.0 | 10.0 | 60000 |
| Percent of Time at No Attach | 24.9% | 43.2% | 0.0% | 100.0% | 60000 |
| Percent of Time at Max Task | 0.0% | 0.0% | 0.0% | 0.0% | 60000 |
| Percent of Time at Short-on-Storage | 24.9% | 43.2% | 0.0% | 100.0% | 60000 |
| Queued Tasks | 0.8 | 1.0 | 0.0 | 5.0 | 60000 |
| Percent of Time at CICS Idle | 10.9% | 31.1% | 0.0% | 100.0% | 60000 |

TABLE 4.9

COLCT-TYPE STATISTICS

| | MEAN | STD.DEV. | MIN | MAX | OBS |
|---|---|---|---|---|---|
| Total Time | 1074.6 | 915.2 | 166.0 | 6027.1 | 231 |
| Wait Time | 927.6 | 860.0 | 112.9 | 5629.2 | 231 |
| Core Storage Usage | 17.8 | 6.5 | 7.1 | 42.5 | 5831 |

TMST-TYPE STATISTICS

| | MEAN | STD.DEV. | MIN | MAX | OBS |
|---|---|---|---|---|---|
| Percent Program Loader Active | 19.4% | 39.5% | 0.0% | 100.0% | 60000 |
| Active Tasks | 4.1 | 4.3 | 0.0 | 17.0 | 60000 |
| Percent of Time at No Attach | 2.6% | 16.0% | 0.0% | 100.0% | 60000 |
| Percent of Time at Max Task | 0.0% | 0.0% | 0.0% | 0.0% | 60000 |
| Percent of Time at Short-on-Storage | 2.6% | 16.0% | 0.0% | 100.0% | 60000 |
| Queued Tasks | 1.0 | 1.1 | 0.0 | 7.0 | 60000 |
| Percent of Time at CICS Idle | 3.2% | 17.7% | 0.0% | 100.0% | 60000 |

TABLE 4.10

COLCT-TYPE STATISTICS

| | MEAN | STD.DEV. | MIN | MAX | OBS |
|---|---|---|---|---|---|
| Total Time | 2432.0 | 1964.0 | 261.6 | 11636.4 | 224 |
| Wait Time | 2326.4 | 1891.0 | 217.4 | 11153.9 | 224 |
| Core Storage Usage | 29.8 | 11.6 | 7.2 | 62.6 | 5964 |

TMST-TYPE STATISTICS

| | MEAN | STD.DEV. | MIN | MAX | OBS |
|---|---|---|---|---|---|
| Percent Program Loader Active | 18.0% | 38.4% | 0.0% | 100.0% | 60000 |
| Active Tasks | 9.7 | 8.5 | 1.0 | 25.0 | 60000 |
| Percent of Time at No Attach | 4.8% | 21.3% | 0.0% | 100.0% | 60000 |
| Percent of Time at Max Task | 2.9% | 16.8% | 0.0% | 100.0% | 60000 |
| Percent of Time at Short-on-Storage | 1.8% | 13.4% | 0.0% | 100.0% | 60000 |
| Queued Tasks | 1.5 | 1.6 | 0.0 | 8.0 | 60000 |
| Percent of Time at CICS Idle | 0.7% | 8.3% | 0.0% | 100.0% | 60000 |

# CHAPTER 5

## 5.1 CONCLUSIONS AND RECOMMENDATIONS

The model has been used to derive certain basic characteristics of the CICS system as seen in Chapter 4. What at first may have seemed to be an isolated problem of lack of core storage turned out to be a complex intertwining of relationships between various components of CICS. The one absolute problem which was discovered was that of the program loader. The program loader routine as implemented by IBM at times performed unnecessary work when loading programs written in a high-level language such as PL/I or COBOL. A strong recommendation as a result of this observation would be to revise those routines which are involved in program loading to bypass the unnecessary operations.

An interesting observation made during the above study is that it is absolutely necessary to place some realistic constraints on the amount of core storage available and on the maximum allowed number of tasks in the system. This is necessary because it has been shown during the simulation study that the throughput of the system and the average response time will reach optimum figures and any further increases in the core storage or task limit will actually start to produce system degradation. This occurs because the increasing number of tasks active in the system at any one time cause larger queues to be formed for the other resources of the system and the average wait time for the tasks increases. However, using this model it is not possible to

derive absolute figures for these two parameters. It would be necessary for any installation interested in these parameters to accurately represent their configuration in the model in order to derive them. The recommendation can be made, however, that in lieu of performing this type of study, an improvement in system performance could very well be obtained by reducing the core storage available and/or the maximum allowed number of tasks.

## 5.2 SUMMARY

It has been demonstrated that the use of this model is a viable tool in solving problems relating to a CICS installation, and as such satisfied the stated objective of this paper. It must be kept in mind that the model will not supply the user with all the answers; knowledge of CICS is a necessity and the ability to interpret the results is a must.

## 5.3 AREAS FOR FUTURE STUDY

A model of this type enables a user to get an understanding of a complex system and its inter-relationships, other than that for which he has intuitive feelings. This is invaluable in problem solving and planning for future revisions. However, for some questions concerning the functioning of CICS, it may be of much more value to be able to derive exact quantitative results rather than only proportional data. To achieve a model of this type, many changes would have to be made to the existing model in four primary areas:

1. A facility would have to be developed within CICS itself

to provide data which could be used to "drive" the model. This data would probably be necessary whenever any change of state occurred in CICS and would have to at least include any necessary parameters which accompanied this change of state. Also, CICS would have to be modified to provide much more detailed statistics than are now available which would be used as input parameters or constants within the model.

2. All of the data available in the system tables would have to be made available to the model in some form.
3. Revisions to the model would have to be made so that it is "driven" by the trace data provided by CICS. It is also conceivable that areas within the model would have to be done in greater detail to support this new scheme of operation.
4. Attention would have to be given to the hardware configuration of the system and in particular to those areas of the model involving input/output operations.

To refine the model to this extent would require considerable effort, but would open up new areas of use.

One additional area which deserves some consideration is the effect of running CICS in a multiprogramming environment. This would introduce "noise" into the model in many areas, and should be considered insomuch as it affects the CICS system. For example, suppose an input/output request external to CICS is

tying up some facility needed by CICS. This should be observed so that measures can be taken to relieve the contention. This enhancement would also be quite extensive, but would improve the effectiveness and usefulness of the model.

BIBLIOGRAPHY


1. Martin, J., Design of Real-time Computer Systems, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1967.

2. International Business Machines Corp., Customer Information Control System (CICS) General Information Manual, IBM, Publication No. GH20-1028-2, White Plains, New York, 1972.

3. Baafi, R. K., A Simulation Model of the Control Data 6400 SCOPE Operating System, Masters Thesis, Lehigh University, 1973.

4. Shulman, F. D., "Hardware Measurement Device for IBM System/360 Time-sharing Evaluation", Proceedings of the Association for Computing Machinery National Conference, 1967, pp. 103-110.

5. Oakes, D. A., "Teleprocessing Systems Design and Design Considerations", Unpublished IBM Internal Document, Toronto, Ontario, 1968.

6. Maisel, H., and Gnugnoli, G., Simulation of Discrete Stochastic Systems, Science Research Associates, Inc., Chicago, 1972.

7. Chow, J. V., "What you need to know about DBMS---Part 1," Journal of Systems Management, Vol. 26, No. 5, May 1975, pp. 22-29.

8. International Business Machines Corp., Customer Information Control System (CICS) Application Programmer's Reference Manual, IBM, Publication No. SH20-1047-3, White Plains, New York, 1972.

9. International Business Machines Corp., Customer Information Control System (CICS) System Programmer's Reference Manual, IBM, Publication No. SH20-1043-3, White Plains, New York, 1972.

10. Martin, F. F., Computer Modeling and Simulation, John Wiley & Sons, Inc., New York, 1968.

11. Martin, J., Telecommunications and the Computer, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1969.

12. Pritsker, A. A. B., and Kiviat, P. J., Simulation with GASP II, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1969.

13. Reardon, R. S., "The Problem of the Peak in Real-time Systems", Unpublished IBM Internal Document, London, England, 1966.

14. Sayers, A., Operating Systems Survey, Auerbach Publishers, New York, 1971.

15. Schmidt, J. W., and Taylor, R. E., Simulation and Analysis of Industrial Systems, Richard D. Irwin, Inc., Georgetown, Ontario, 1970.

16. Sippl, C. J., Computer Dictionary and Handbook, Howard R. Sams & Co., Inc., Indianapolis, 1966.

17. Bourne, C. P. and Donald F. Ford, "Cost Analysis and Simulation Procedures for the Evaluation of Large Information Systems", American Documentation, Vol. 15, No. 2, 1964, pp. 142-149.

18. Forrester, Jay W., Industrial Dynamics, The M.I.T. Press, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1961.

19. Vaucher, Jean G. and Pierre Duval, "A Comparison of Simulation Event List Algorithms", Communications of the ACM, Vol. 18, No. 4, April 1975, pp. 223-230.

20. Bard, Y., "Performance Criteria and Measurement for a Time-Sharing System", IBM Systems Journal, Vol. 10, No. 3, 1971, pp. 193-216.

21. Drummond, M. E., "A Perspective on System Performance Evaluation", IBM Systems Journal, Vol. 8, No. 4, 1969, pp. 252-263.

22. Seamon, P. H. and R. C. Soucy, "Simulating Operating Systems", IBM Systems Journal, Vol. 8, No. 4, 1969, pp. 264-279.

23. Cheng, P. S., "Trace-Driven System Modeling", IBM Systems Journal, Vol. 8, No. 4, 1969, pp. 280-289.

24. International Business Machines Corp., Customer Information Control System Program Description Manual, IBM, Publication No. H20-0605-0, White Plains, New York, 1969.

25. Seamon, P. H., "On Teleprocessing System Design - Part VI - The Role of Digital Simulation", IBM Systems Journal, Vol. 5, No. 3, 1966.

26. Sherman, S., et.al., "Trace-Driven Modeling and Analysis of CPU Scheduling in a Multiprogramming System", Communications of the ACM, Vol. 15, No. 12, December 1972, pp. 1063-1069.

27. Stimler, S. and K. A. Brong, "A Methodology for Calculating and Optimizing Real-Time System Performance", Communications of the ACM, Vol. 11, No. 7, July 1968, pp. 509-516.

# APPENDIX A

Following is a brief discussion of the GASP2 internal files used in the model. For each file the title is given, the sequence and sequence element and a description of the file elements.

FILE 1: Event Queue
Ascending sequence on element 1

Element 1: Event time
Element 2: Event code

FILE 2: Allocated Storage Queue
Ascending sequence on element 1

Element 1: Owner identification
Element 2: Beginning allocation address
Element 3: Length of storage request
Element 4: Storage type
Element 5: Transaction number

FILE 3: Suspended Task Queue
Ascending sequence on element 3

Element 1: Transaction number
Element 2: Length of storage request if suspended for short-on-storage
Element 3: Time into queue
Element 4: Suspend code

FILE 4: Active Task Queue
Descending sequence on element 2

Element 1: Transaction number
Element 2: Priority of the transaction

FILE 5: Fragmented Free Storage Queue
Ascending sequence on element 1

Element 1: Storage address
Element 2: Length of free area
Element 3: Storage type

FILE 6: Program Loader Backlog Queue
Ascending sequence on element 1

Element 1: Time entered into the queue
Element 2: Transaction number
Element 3: Program to be loaded

FILE 7: Exclusive Control Record Queue
Ascending sequence on element 1

Element 1: Transaction number
Element 2: User data base file number
Element 3: Storage address of record

# APPENDIX B

In preparing this simulation model it was necessary to make some underlying assumptions in order to limit the scope of the problem to a model of CICS. Also, some assumptions were necessary in order to obviate the need of becoming involved in trivialities. The primary assumption was that only those problems which were directly relatable to the CICS environment were studied. No problems which indirectly affected the system were considered. For example, it was assumed that CICS was being executed in a dedicated environment; that is, one in which it was the only user job being queued for the resources of the system. This assumption was necessary because an unfavorable job mix in a multiprogramming environment could itself cause a severe degradation in response time, even though there was nothing wrong with the CICS operation itself. The solution to a problem such as this is completely trivial. It would be to create as favorable a job mix as possible, and running stand-alone would be the most favorable job mix. Also, it was assumed that all resources were operating with no mechanical problems. It was again obvious that a loss of a channel or a direct-access storage facility would result in longer queues being formed to use the remaining facilities, and this in itself would cause a longer response time. One final assumption was made, that all code in all programs, either user-written application programs or IBM-written

control programs, was efficient, "good" code and that it made optimum use of core storage and other resources. This is not to say that the logic behind the code is pure, but that the code is. An attempt at exposing problems of a coding nature are not within the scope of this paper, and in themselves are not even worthy of extensive research to discover them.

# APPENDIX C

From the author's standpoint it was deemed imperative that the GASP2 routines be rewritten from FORTRAN into PL/I. The author has much more experience using PL/I than FORTRAN, the installation where all of the development and testing work was done for the model offered much greater benefits to the PL/I user than the FORTRAN user, and the PL/I language itself offered some features which simplified the programming and made it a more viable solution than if it had been done in FORTRAN.

The initial step was to transcribe the existing routines from FORTRAN to PL/I. Primary concern was given to exact representation of the reproduced code, and to determine the best variable type for each scalar and array, since PL/I has available several more data types than the INTEGER and REAL types which FORTRAN employs.

The next step was to make the appropriate revisions to all routines that were necessary to transform GASP2 to GASP2A as described in reference [12]. GASP2A differs from GASP2 in the fact that the filing array in GASP2A is floating point and pointers for the filing array are in a different fixed-point array, while GASP2 only employs one fixed-point array which is used to store program data and pointers for this data. With GASP2 there was always the potential problem of truncation because a scaling factor had to be applied to each element before it was entered to the array. One minor difference between the

model's version of GASP2A and that discussed in reference [12] is that both integer and real values are stored in the same array in the model's version, where as in Pritsker's version from reference [12], integer values are stored in the array containing the pointers. The major reason for this is that Pritsker was attempting to conserve on the use of core storage, since in some computers floating point variables require more core than do fixed point variables. However, in the preparation of the model, core storage was never a restriction, and the benefit accrued by having all file variables in a single filing array was considered a desirable benefit.

The third and final step of the translation of a FORTRAN-based GASP2 to a PL/I-based GASP2 was to add programming logic to take advantage of several options available in PL/I and to tailor the routines to meet some of the specifications of the model and to provide for easier program testing. These changes follow:

1. All arrays were DECLARED with a variable for the dimension, and were given the CONTROLLED attribute. This enables these arrays to be dynamically allocated during program execution time by use of the ALLOCATE statement. Thus, by reading the dimensions of these arrays on a data card, the size of the arrays can be varied without recompiling the model.
2. The double-dimensioned array which is used to store the filing elements was transposed so that the number of rows

is the total number of filing elements in the array and number of columns is the number of entries for each filing element. This was desirable because PL/I stores double-dimension arrays in row major order, while FORTRAN stores double-dimensioned arrays in column major order.

3. Subroutine SET was modified in the routine which adds elements to highest-value-first (HVF) files. Previously, if the ranking value of the row being added to the file was equal to the ranking value of the row being tested in the file, then the new row was added ahead of the tested row. This logic was revised to add the new row following all current rows which have equivalent ranking elements.
4. Subroutine MONTR was revised to give a third type of potential monitoring information. When an event code greater than 200 is encountered, MONTR calls subroutine SUMRY which prints out all generated data, time generated data and filing arrays. Also, MONTR was revised when handling the case where the event code is between 100 and 200. Previously, the subroutine would have printed out the entire filing array. This was judged to serve no useful purpose, and was wasteful of both time and paper. Thus, the routine now only prints out that portion of the filing array which is "active". Starting from the end of the array, all elements are tested for a non-zero value. The first such row encountered would be the

delimiting point of the printout.

Once the translation had been completed, the task still remained of verifying the equivalence of the PL/I version of GASP2 to the FORTRAN version. This verification step took two forms. The most obvious form was to run an identical model using both versions of the simulator, and then to compare the output. This was done with several of the example models in reference [ 12 ] . Secondly, all of this testing was done in a time-sharing environment using the PL/I Checkout Compiler. The PL/I Checkout Compiler is an interpretative type compiler written by IBM primarily to be used in interactive testing sessions. By using various facilities of this compiler, values of any or all variables could be displayed whenever they were changed, the flow of the execution could be observed as a trace of statement numbers was printed or the execution of the program could be temporarily halted to check on or change the values of variables. By testing in this manner, it was possible to quickly duplicate the results of the FORTRAN version while using the PL/I version.

# APPENDIX D

The CICS model which has been developed consists of 38 user-written subroutines in addition to 14 GASP2 system subroutines, the user-written event selection subroutine and the user-written OTPUT routine. There are three input files to the model with the following functions:

1. The standard GASP2 input cards containing run control data, parameter data, file data and initial events.
2. The file containing the initialization data for the Processing Program, Program Control and File Control tables.
3. The file containing variable parameter data for the system, such as the number of core storage blocks to be allocated, the mean time between transaction arrivals and so forth.

These files will be referred to throughput this discussion as the GASP2 file, Initialization file and Parameter file, respectively. There are also seven internal "files" which are stored in the GASP2 filing array QSET. These "files" are discussed more thoroughly in Appendix A.

Perhaps the best way to explain this complex set of procedures is to discuss the routines as they might be encountered during a normal run of the model.

## SYSTEM INITIALIZATION

When CICS is to be brought up from a 'cold start', it is

necessary for several initialization procedures to be performed. One of these procedures is to load the system tables and those programs which are marked as being permanently resident. The function of subroutine SYSINIT is to obtain core storage for these tables and programs, and also to obtain core storage for a standard system area used by CICS. Naturally, the amount of core storage obtained for the tables is dependent on the number of entries in each table. The number of entries for each table is one of the parameters entered in the Parameter file. A call to this subroutine must be the first initial event entered into File One from subroutine DATAN. After CICS has been initialized, SYSINIT branches to Terminal Control to commence polling of the terminals for activity.

TERMINAL CONTROL

In the model the action of Terminal Control is represented by two subroutines, TC_NEXT and TC_GET. TC_NEXT determines the time of the next Terminal Control read; that is, when, through polling, a terminal was found requesting activity. The subroutine uses an exponential distribution to determine the interval to the next read request, with the mean time between requests, XMU_ARRVL, being entered as one of the parameters on the Parameter file. TC_NEXT also determine which transaction type is being requested by taking a random sample from a uniform distribution of transaction numbers. Then an event to call TC_GET is entered to File One for the generated next read. Since only

one task can be attached to an individual terminal at any one time, TC_NEXT locks itself out whenever the condition arises that all terminals have an active task. The routine is unlocked whenever one of the tasks is terminated. To initially call TC_NEXT, it is necessary for the second initial event in the GASP2 file to be a request for this subroutine.

TC_GET simulates the actions necessary to perform a read from a terminal. The first thing that it does is to make a conditional request for a block of core storage equal to the message length from the terminal. Since the system being studied uses video tubes exclusively, the message length was set equal to the size of the screen image, 480 bytes. However, the model could easily be revised to handle other terminal models or configurations with multiple types of terminals. Since the request for core storage was conditional, a short-on-storage condition will cause the terminal that is requesting the action to remain in a pending status. Initialization of the terminal event will again be attempted in the next polling loop. However, if the storage request was successful, TC_GET will initiate the I/O event to read the input from the terminal. At this time, TC_GET will go into a wait state on this operation; that is, no more action can be done for this terminal until the input event is completed. Sometime into the future, the input event will be completed. Now, when TC_GET regains control, it confirms that the read was error free, translates the input to internal machine code and releases

the core storage which it had initially obtained for this terminal. The final action is to "attach" a task within CICS to process the request from the terminal. TC_GET simulates this by inserting an event into File One with the event time being equal to the current time and the event code being that of the Task Control Attach subroutine.

TASK CONTROL

Task Control consists of eight routines which simulate the actions of CICS at the task level. Among these actions are:

1. Attach a new task.
2. Suspend an active task.
3. Place a task into a wait state until completion of a pending event.
4. Resume a task that has been in a wait state.
5. Change the priority of an active task.
6. Test the CICS system for the availability of resources.
7. Dispatch a task which is not suspended or waiting.
8. Terminate an existing task.

As mentioned previously, the first action to be taken with a new task by Task Control is to attach the task. This entails verification of the Task ID, obtaining core storage for a task control area (TCA) and task work area (TWA), placing the task into the active task queue and loading the initial program to be used by the task if it is not yet resident in core storage. Subroutine KC_A simulates these actions. An additional function performed by IC_A is to determine if the condition of maximum allowed number of tasks has occurred. The maximum number of tasks allowed to be attached at any point in time is a value read from the Parameter file. If this condition has occurred, then no

more new tasks are attached until an existing task has terminated.

While processing, a task may encounter certain conditions which prohibit it from processing further. Among these conditions are:

1. A storage request was made but not enough core storage is available.
2. A request was made to load a program module or table and the loader routine was already servicing another task.
3. An error has occurred in a task and the task is attempting to "dump" out to a file; however, another task is already using the dump resource.
4. A request was made for temporary storage, either internal or external, but not enough was available.

Under any of these circumstances, the task is suspended and placed in a non-active state until the prohibiting condition has abated. Subroutine KC_S of Task Control is used to simulate these actions. The routine finds the appropriate entry in the active task queue and places a copy of the entry into a suspend queue. This queue is ordered by the time into the queue so that if multiple tasks are suspended for the same reason, then the task suspended for the longest period of time will be re-started first. The routine also places an indicator into the Task Control Area (TCA) to indicate that the task can no longer be dispatched.

A similar condition to being suspended is being placed into a wait state. Here the task is not being delayed because of some external condition which is affecting it, but because of some task-related event which is pending until some time in the future. The most familiar reason for waiting is an outstanding input/output event. Once the input/output event has been initiated by

the task, it is placed into a wait state until the actual physical actions have taken place to either read or write the record. This sequence of events is necessary if a multiprogramming/multiprocessing environment is to be maintained. Subroutine KC_W simulates these actions in much the same way that subroutine KC_S simulates the suspension of a task, except for the following exceptions:

1. Instead of indicating in the TCA that the task is suspended, subroutine KC_W indicates that it is waiting for a pending event.
2. An entry is placed into a list of tasks which are currently waiting. In the event that the condition arises that there are no tasks which can be dispatched, CICS will return control to the operating system for a maximum of two seconds. Whenever one of the tasks in the list has its pending event completed or the two seconds has elapsed, the operating system will again return control to CICS.

Whenever all the pending events for a task have been completed, it is necessary to remove the task from the suspended task queue and indicate that the task is again an active dispatchable task. Subroutine KC_R performs this function. A search is made of the suspended task queue to find the task, and when found it is removed from the queue. Also, the indicator in the TCA that shows that the task is waiting is turned off. Finally, control is returned to the task for further processing.

At times during the processing of a task, it is advantageous for CICS to dynamically change the priority of the task. One particular instance is when that task is using the loader. Since only one task can make use of the loader at any one time, it

would be beneficial for this task to make use of the loader and release it in as short a time as possible. This is accomplished by giving the task the highest possible priority while it is using the loader, so that it will always be the first task to be dispatched if it is not suspended or waiting. Routine KC_C simulates this by removing the task from the active task queue, revising the priority and re-inserting the task back into the queue. Again, an indicator is turned on to show that the task has had its original priority changed.

One of the main focal points of the CICS system is the Task Dispatch routine. This routine is responsible for selecting the task that has the highest priority and which is not suspended or waiting and to give that task use of the CPU; that is, to either start or resume execution of the task. Routine DSPTCHR performs this function by searching through the active task queue for a task which meets the above criteria. If none are found, then the dispatched issues a wait and control return to the operating system, as described above. While stepping through the active task queue, the routine examines the dispatching indicator for each task. If it indicates that the task is active, control is transferred directly to the task. If the indicator says the task is waiting for a pending event, but there are no more pending events outstanding for this task, then the dispatched will branch to routine KC_R and resume execution of the task. If the indicator shows that the task is suspended, the dispatcher examines

the suspend code and tries to determine if the task can be re-started. If the task was suspended because of an inadequate amount of core storage available, then the dispatched determines whether there is enough core storage available at the present time. If there is, the dispatcher returns control to the task at the point where it was suspended; that is, where the task was requesting core storage. If the task was suspended for any other reason, the dispatcher then increments to the next task and attempts to dispatch it.

The last function which can be performed upon a task is for it to be terminated. After every task has completed processing, either normally or abnormally, the system branches to routine KC_T to perform task termination. Subroutine KC_T is responsible for releasing all resources held by the task while it was active. These resources included both task-related and terminal-related core storage, and any temporary storage which may have been acquired. Also, KC_T collects statistics on the task such as the total time in the system and the total time spent waiting by the task. Its last function is to determine if the system was at an impasse' due to having reached the maximum allowed number of tasks in the system. If this was the case, then KC_T turns off the maximum task indicator which tells Terminal Control that additional tasks may now be read in.

PROGRAM CONTROL

The routines of Program Control work within the system at

the next lower level below Task Control. This hierarchical structure is necessary because a single task may use multiple programs and also because a single program may be used by multiple tasks at any point in time. The functions performed by the five Program Control subroutines modeled within the system are:

1. Load a program/module from external storage.
2. Link from the currently-executing program to a lower program.
3. Return from the currently-executing program to a program at the next highest level.
4. Transfer control from the currently-executing program to one at the same level.
5. Delete a program/module from being resident in core storage.

Obviously, before a program can be executed, it must be resident in core storage. This implies that before a task can perform its function, the initial program used by that task must be resident in core storage. It is the function of the Program Control Fetch routine to ensure that a requested program is loaded into core storage, if necessary, and of subroutine PC_F to model this routine. When a task is ATTACHED, the name of the initial program to be used by the task is placed in the TCA, and the address of the Program Control Fetch routine is stored in the TCA as the address to which the Task Dispatcher will transfer control whenever the task is dispatched. When routine PC_F is entered, the first action to be performed is a search of the Processing Program Table (PPT) to determine if the program is already resident. If it is not, then PC_F changes the priority

of the task so that it has the highest possible priority. This helps to minimize the time spent by the task in PC_F. Then the routine reads a series of control blocks, dictionary blocks and finally text blocks until the program is loaded into core storage. Once it is loaded, the routine indicates in the PPT that the program is now resident, and returns the task to its original priority. Since the function of loading a program is relatively slow, a queue of load requests can easily develop while another request is being processed. Thus, after the active request has been processed, PC_F searches the queued tasks to determine if any have been suspended because PC_F was not available. If a task has been suspended, then PC_F removes it from the suspend queue and raises its priority to the maximum. Thus, the next task to be dispatched will be this request for the program loader. When PC_F determines that the program is loaded (either from a previous use of the loader or from the current use), it concludes with one of the following actions:

1. It branches to the program and commences execution.
2. It returns to the program which issued the request for the loader.

The latter alternative only occurs when the program being loaded has a status of load-only. In this way it is possible for a program to dynamically load tables or other data needed for its operation.

Since a task may use more than one program in providing its service, CICS must provide a means of transferring from one

program to another. Two Program Control routines provide this function in two different ways. Control can either be transferred from one program to the next, or it can be linked from one to the next. Transfer of control will be discussed first, followed by linking.

If control is transferred the system branches from a program to another one at the same logical level, and the ability of returning to the first program by simply ending the branched-to program is relinguished. When the first program relinguishes control, it must be released from the task, a function modeled by subroutine PC_R. PC_R releases any core storage which was obtained by, and used by, the program. It also reduces a count in the PPT indicating the number of users of a program at any point in time. After the first program is released, control is transferred to the second program by using routine PC_F. If this program would call no others, then at its end control would return to the task control routines and the task would terminate.

Linking from one program to another means that the system branches from the currently executing program to one at a lower logical level, and it maintains the information necessary to return to the calling program in an area called the register storage area. After saving this necessary information, the second program is initiated by using routine PC_F, without first releasing the initial program. However, when the second program has completed and been released, control is not returned to CICS but

to the calling program. This type of control transfer is generally used when a program calls a routine to perform some generalized common function, and is modeled by subroutine PC_L.

A final function performed by Program Control is to delete a program or module from residency in core storage. Normally, all programs remain core resident as long as the CICS system is not short on storage. However, the user has the option of dynamically deleting a program, possibly because it is unusually large, or because it does not have a high frequency of use. Subroutine PC_D simulates this by checking the PPT to determine if there are any users of the program or if the program is marked as permanently resident. If either one of these conditions holds true, then the program cannot be deleted and PC_D ends. However, if it can be deleted, then PC_D frees the core storage used by the program and then marks the program as non-resident in the PPT.

STORAGE CONTROL

The Storage Control routines have, perhaps, the most far reaching impact on the entire CICS system, since, along with the CPU, core storage is one of the most precious commodities of a computing system. In the use of CICS, every effort should be placed upon judicious use of this commodity, both by internal CICS routines and also by user-written routines. Thus, although they comprise only three of the 38 user routines of this model, they are logically the most complex and extensive. Throughout this discussion it will be necessary to keep in mind that IBM's

System/360 and System/370 computers allocate core storage in blocks of 2048 bytes of 8 bits each. That storage is then subdivided into smaller segments, as required, when needed by the using routines.

The routine which simulates obtaining core storage operates under a first fit criterion. It first attempts to find one of the 2048-byte blocks from which storage has already been allocated. If it is successful in this, and there is sufficient remaining storage in that block to satisfy the current request, Storage Control Obtain (SC_O) allocates the needed storage out of that block by updating a storage accounting area and then returns the address of the allocated storage to the requesting task. If, however, there are no 2048-byte blocks from which storage has already been allocated, or if there is not sufficient storage in one of the already-allocated 2048-byte blocks, then SC_O will select an unused 2048-byte block from the pool of blocks made available at system start-up time. The storage request will then be allocated from this block, starting at the low order byte. If the storage request is for greater than 2048 bytes, then adjacent blocks of 2048 bytes are necessary to fill the request.

However, if the request cannot be filled from the storage configuration available to the system at the present time, then SC_O will take measures to attempt to provide sufficient free core storage for the request. The first action it will take will be to free any areas occupied by programs which are

resident in core storage, which are not marked as permanently resident, and which have no current users. It accomplishes this by stepping through the PPT, starting at the end, and finding a program which satisfies the above-mentioned criteria, and then freeing this core storage. It then again attempts to satisfy the request. If the request still cannot be satisfied, SC_O continues up the PPT, freeing programs and testing the request, until the top of the PPT is reached.

If the storage request is yet unsatisfied, SC_O takes one final drastic action; it makes available to CICS a separate area of core storage which was set aside at start-up time. This area is known as the storage cushion. When this happens, SC_O also sets an indicator which prohibits any new tasks from being initiated from the terminals. It is the hope of CICS that the storage cushion can satisfy all the requirements of all the tasks which are currently active, so that those tasks can be terminated, their core storage released to the system and the core storage environment returned to a more normal state of use. Unfortunately, there are times when the storage cushion cannot satisfy all requests being made upon it, and CICS has no final option except to suspend the task which is requesting core storage. An entry is placed into the suspend queue for this task, and an indicator is turned on in the task's TCA indicating that the task is suspended. This task will only be re-started when there is sufficient core storage available to handle its request.

The counterpart of the Storage Control Obtain routine is the Storage Control Release routine, SC_R. Storage Control gives the user and other CICS routines some degree of flexibility in releasing core storage, since core storage, when it is allocated, is tagged as being either task, program or terminal related, and since all storage for a task is chained together and all storage for a terminal is chained together. The user or CICS routine has the option of releasing all storage attached to a specific terminal, all storage owned by a particular task, or any specific block of terminal, program or task related storage which is identified by its storage address. The routine essentially reverses the process performed by the SC_O routine; that is, it removes that block or those blocks which were designated from the allocated storage queue and updates counts on the number of users and number of free bytes in each 2048-byte block. It also updates, if possible, the queue of fragmented free storage blocks or core in an attempt to develop one contiguous block or core storage rather than two or more disjointed blocks. The final action of SC_R is to scan the blocks of storage allocated by the storage cushion to determine if any of its blocks have any allocated storage. If not, the cushion is returned to the system, and the restriction of ATTACHing new tasks is removed.

A third routine in Storage Control is a routine which attempts to remove from the suspend queue those tasks which had been suspended due to insufficient core storage and which can

now be restarted. This routine is represented by subroutine SC_F in the model. The routine searches through the suspend cueue to determine if any tasks are waiting for additional core storage to become available. Once a task is found, SC_F calls routine SC_O in an attempt to obtain the requested storage. If it is obtained the task is again marked as dispatchable and the routine ends. If the storage was not obtained, then SC_F increments to the next suspended task and attempts to do the same thing. If, after processing through the entire suspend queue, no tasks are found which were suspended due to insufficient core storage, then SC_F will attempt to release the storage cushion if it is allocated. If it is allocated but cannot be released, then SC_F will attempt to release programs which are in core but not being used. After a program has been released, an attempt is again made to release the storage cushion, in the hope that the storage released by deleting the program was in the storage cushion. The actual intent of this entire section of logic in CICS is to get the storage cushion released back to CICS so that the restriction on starting new tasks while the storage cushion is allocated can be removed. In effect, it is an attempt to decrease the response time of the system by freeing one of the constraints.

TEMPORARY STORAGE

As mentioned previously Temporary Storage provides for a "scratch pad" to be used by a task, especially if it is long

running or requires more than one set of data to be written to a terminal. CICS, as implemented by IBM, provides for Temporary Storage data areas to be either on a Direct Access Storage Device or in main storage, but the CICS system which is the basis for this model only implemented that part of Temporary Storage which uses Direct Access Storage Device data areas, and hence, that is the only part modeled.

Data is written out to the Temporary Storage file through the Temporary Storage Put routine, TS_P. The routine first determines if there is an available block in the file. If there is none, the task is suspended. If there is an available block, core storage is allocated for the record and the record is written to the file. After the write is complete, TS_P releases the core storage where the data record was constructed. Then, since there is a restriction that there can only be one input/output event pending to the Temporary Storage dataset at any point in time, TS_P searches the suspend queue to determine if any tasks have been suspended because of the inavailability of Temporary Storage. If there is such a task, it is removed from the suspend queue and made dispatchable again. If not, TS_P returns control to CICS.

The routine which retrieves a task's data from the Temporary Storage file is TS_GR, Temporary Storage Get/Release. This routine also releases ownership by a task of a Temporary Storage block, or, in combination, gets the block and then releases it. When the request includes a get from the file, TS_GR must obtain

a core storage area into which the record will be read. Then the routine initiates the read. Once the read has been completed, TS_GR, like TS_P, will restart a task which has been suspended because of the inavailability of Temporary Storage, if there is such a task.

If the request was for a release only, TS_GR releases ownership of the block. If any task was suspended because there were not enough Temporary Storage blocks allocated, then it is given ownership of the block and marked as dispatchable. If the request was for a combination get and release, both sections of applicable logic are performed.

FILE CONTROL

File Control routines are those routines in CICS responsible for all operations involving the user data base. The system models the eight major areas of File Control with the following routines:

1. FC_OCL -- This subroutine is responsible for opening and closing files in the user data base; that is, it creates a linkage between the task and the file to enable input/output operations.
2. FC_S -- This subroutine sets up work areas so that a task may browse through a file; that is, it makes possible for a task to obtain the next logically sequential record from a file upon request.
3. EC_GN -- This subroutine retrieves the next sequential record as set up by a browse operation.
4. FC_RES -- This subroutine resets file work areas to facilitate browsing at a new logical location in the file.
5. FC_GET -- This subroutine performs a direct read upon the user data base.
6. FC_PUT -- This subroutine performs a direct write of a new or updated record to the user data base.

7. FC_GA -- This subroutine allocates a file work area in which a new record can be constructed.
8. FC_RL_E -- This subroutine releases control of a record read with exclusive control or can be used to release file input/output and browse work areas.

The first action that a task must take with the user data base is to issue an open, for unless the file is opened for the task, no operations can be directed towards that file. FC_OCL searches the File Control Table (FCT) for the appropriate file. If it is found, it is indicated as being open in the task's TCA. If it is not found, then an error indicator is returned to the task and the task will abnormally end. A similar set of operations occurs when routine FC_OCL is used to close a file, except that the indicator in the task's TCA is shown as closed.

Four of the remaining seven routines, FC_S, FC_GN, FC_RES and FC_RL_E, are primarily concerned with presenting records to the task in a sequential manner, while the other three, FC_GET, FC_PUT and FC_GA, are concerned with direct operations on the file. At times it may not be possible for a user to uniquely identify a particular record which he wishes to interrogate in a user data base. This may be due to the fact that several records have identical keys. In this case CICS makes it possible for the task to access part or all of the records which have synonymous keys and allows the user to determine which one is the appropriate record. At other times there may be no duplication of keys and CICS will directly access the desired record.

When it is desired to sequentially access a series of records,

it is necessary to obtain three areas of core storage, a file input/output area (FIOA), a file browse work area (FBWA) and a file work area (FWA). This is the duty of routine FC_S. It searches the FCT for the appropriate file ID, and if found, uses lengths stored in the FCT for the file to initialize for the storage requests. If the file is not found in the FCT, control is returned from FC_S with the indication that the file was not found and the task is abnormally ended. FC_S calls Storage Control routine SC_O for each of the three areas. If the storage is not available for any of the three areas, then the calling task is suspended at that point. After all three areas have been successfully obtained, an indicator is turned on in the task's TCA to indicate successful completion of the function.

To retrieve the next (or first) logical record as specified by a generic or specific key (a generic key is one where only the high order portion is assigned and the low order portion is zeroes or blanks), the system uses subroutine FC_GN. The routine first verifies that subroutine FC_S has been previously executed for this task/file combination. Then, if this is the first sequential read, the file is unblocked, or the end of a physical block has been reached, the routine issues a read to the file. However, if a physical blocked record is available in the FWA and the end of the block has not yet been reached, then subroutine FC_GN will only de-block the next logical record and present it to the task. Finally, after retrieving the logical record, FC_GN releases the

storage occupied by the FIOA by using Storage Control routine SC_R.

Subroutine FC_RL_E is provided by CICS to perform two basic functions:

1. To release all input/output and work areas associated with a task/file combination.
2. To release all exclusive control attributes for a specified task/file combination.

The latter function does not apply for sequential accessing of records, and its discussion will be deferred until later. However, the first function is applicable to the browse operation. This routine supplies an easy method of releasing the core storage allocated by routine FC_S for the FIOA, FBWA and FWA. Again, it first verifies that the file has had a browse operation initiated for it by the task. If so verified, it then uses Storage Control routine SC_R to release all three areas.

It may occasionally be desirable for a task to end sequential processing at one point on a file and resume sequential processing at a different logical record. One way of doing this would be to call routine FC_RL_E followed by another call to routine FC_S for the new logical key. However, to minimize system overhead, a routine, FC_RES, is provided to perform the same function. All that is really necessary for the desired operation is to release the current FWA and to obtain a new FWA pertinent to the new logical request. After first verifying that the task has initiated this file for browsing, subroutine FC_RES performs a Storage Control release (SC_R) for the existing FWA and allocates

a new one by using the Storage Control Obtain (SC_O) routine.

If a record can be uniquely identified by its key, it would be desirable for the task to retrieve that record directly, for there would normally be much less overhead involved. The model uses subroutine FC_GET to simulate the direct reading of a logical record. CICS provides for two modes of direct reading:

1. Read-only, where a record is accessed and can only be used for inquiry purposes, and
2. read-for-update, where the record is read with the intention of updating some field or fields in the record and then putting the updated version of the record out to the file again.

In order to use the latter mode, it is necessary for the task to have exclusive control of the record. This means that no other task may access this record for update until it has been rewritten to the file or the exclusive control has been removed. This other function of subroutine FC_RL_E was referred to above. It will release exclusive control of all records for a specified task/file. This would be necessary if the task never rewrote the records that it read, possibly because it abnormally ended or for some other reason. Subroutine FC_GET first verifies that the file is opened for either input or update. If so, the routine uses Storage Control SC_O to obtain a FIOA into which the record is read. If the record is being read with exclusive control, an area (an FWA) is also obtained into which the record will be queued. FC_GET then initiates the read operation. After the appropriate record has been read, FC_GET releases the core

storage obtained for the FIOA only if core storage had been obtained for a FWA. Otherwise, the record is returned to the user in the FIOA. Finally, the routine updates some statistics and ends.

The counterpart of the routine to directly read a record is the one which directly writes a record (FC_PUT). This routine is used to both add new records to a file and to rewrite a record which had previously been read by routine FC_GET. FC_PUT finds the correct entry in the FCT for this task's file. If the task is attempting to write a new record to the file, the routine will verify whether the file can accept new records by interrogating the FCT entry. Also, it verifies that the file has been opened by the task for output or update. If any one of the above conditions is not met, the write is terminated and the task is abnormally ended. If everything checks out with FCT entry, FC_PUT will initiate the write operation. When the write operation has been completed, FC_PUT uses the Storage Control Release (SC_R) routine to deallocate the core storage for the output area. Alos, if the record had been obtained with exclusive control, the queue element for the record is freed. Again, as in FC_GET, FC_PUT collects some statistics and then ends.

Before a new record can be written to an output file, it is necessary for the task to obtain an area of core storage in which the record will be created. The task cannot directly use the Storage Control Obtain (SC_O) routine because it is necessary for

File Control to be able to access information from the first 11 bytes of this area which are pertinent to the write operation. For this reason the model provides a routine (FC_GA) to get an area of core storage in which the output record is created. The routine, as in all other File Control routines, searches for the correct entry in the FCT and abnormally ends the task if it is not found. FC_GA then uses the information coded in the FCT entry to obtain the proper length work area. As in all other routines which use SC_O to obtain core storage, if the core storage is not obtained, the task is suspended. If the storage is obtained, FC_GA ends normally.

MISCELLANEOUS ROUTINES

Occasionally while processing, a task or CICS control routine may encounter a condition which prevents it from accomplishing its designated duty. When this happens, it is highly desirable that the program problem can be determined, and if possible, eventually fixed. CICS provides the ability to list all or portions of the core storage associated with a task as an assist in determining the cause of the trouble. In the model this function is represented by the routine DMPCNTL. DMPCNTL determines which areas of core storage are to be dumped and writes images of them out to a sequential file. To simplify the dumping operation, DMPCNTL operates as a serially reusable resource so that all of the core image records for a particular task appear consecutively on the file. Since it is serially reusable, only

one task may be active in DMPCNTL at any point in time. If another task enters DMPCNTL while it is active, the second task must be suspended pending completion of the dumping of the active task. If it is not active when entered by a task, DMPCNTL determines which portions of the task's storage are to be dumped. Then, preceding the writing out of each area of core storage, DMPCNTL writes out a header identification record. After finishing dumping all requested areas of core storage for the present active task, DMPCNTL interrogates the suspended task queue for any tasks which may be awaiting its services. If a task is found, it is removed from the suspended task queue and marked as being dispatchable. Also, DMPCNTL is again marked as being active so that the currently-restored task is assured of getting control.

The demand for the services of CICS is not constant throughout the period of time that it is active. In fact there may be times when there is an extended lull of activity. In order to take full benefit of the operating system's multiprogramming capabilities and to use the computing system to its fullest, CICS can relinguish control back to the operating system for a specified period of time or until some component of CICS requests control again. If, after stepping through the entire active task queue, no task is found by the DSPTCHR routine which can be initiated, the model branches to routine OS_WAIT. This routine puts an event into the Event queue which will be executed at the current time plus two seconds. Also, it places an entry into a list of

tasks which are waiting upon a pending event. If any task in the list has its pending event satisfied while control is not with CICS, it will again be given control.

Several times throughout the above discussion the initiation and return from input/output events has been alluded to. As in other multiprogramming systems there is a continuing interaction for all the resources of the computing system, especially the central processing unit (CPU) and the input/output channels. One method of controlling the sharing of these resources, the method employed by the operating system on the IBM System/360 and System/370, is the use of interrupts. For instance, when a program wants to perform an input,output operation, it essentially only informs the operating system of its intentions rather than performing the input,output action itself. When the operating system is aware of the program's intention, it interrupts the program so that it no longer has control of the CPU. It then schedules the input/output event with the channel. From this point on the channel controls the operation. Upon completion of an input/output event, it notifies the operating system and the program is marked as being dispatchable.

The model uses two subroutines to simulate the above actions. Routine READWRT schedules the completion time of the input/output event, while routine OS_POST receives the notification that a pending event has been completed and posts the task as being dispatchable again. READWRT uses an algorithm developed in

reference 5 to determine the time when an input/output event is to be completed. The algorithm states that the total time for a file event is the sum of the seek time, command transfer time, data transfer time, rotational delay and average wait time for the channel. Seek time is the time required to position the read/write heads of the disk drive at the correct cylinder. Command transfer time is the time taken to transfer the appropriate channel commands for the input/output event from core storage to the channel. Data transfer time is the length of time needed to move the data from the disk to core storage or from core storage to the disk. Rotational delay is the time for the rotating disk to spin so that the appropriate record is under the read/write head. The average wait time for the channel is a function of the probability that the channel is busy, the average service time per file event and an interference factor based on the utilization of all disk arms available to the channel. Subroutine READWRY uses the algorithm to calculate the elapsed time for the input/output event based on the access times for an IBM 3330 type disk storage unit. Once the elapsed time has been calculated, the routine adds it to the current time TNOW and inserts an event into File One to indicate the end of the file event. The event is used to initiate routine OS_POST.

As mentioned previously subroutine OS_POST is used to indicate the completion of a pending event, whether the event is associated with a task or with a CICS routine. It is also to

this routine that the operating system returns control if CICS itself has been in a wait state due to inactivity. If the event is a terminal write, then the terminal's entry in the TCT is updated to indicate that there are no pending events outstanding for the terminal. Also, the task which issued the Terminal Control write is removed from the suspend queue and is marked as being dispatchable again. If the event is not associated with a terminal, then the pending event counter for the task is decremented by one and the task is removed from the list of tasks waiting for the completion of a pending event.

As required by GASP2 a routine to call the programmer's events is needed. In the model this routine is used to not only call the requested routine, but to also provide the logic to simulate the flow of control through various representative tasks.

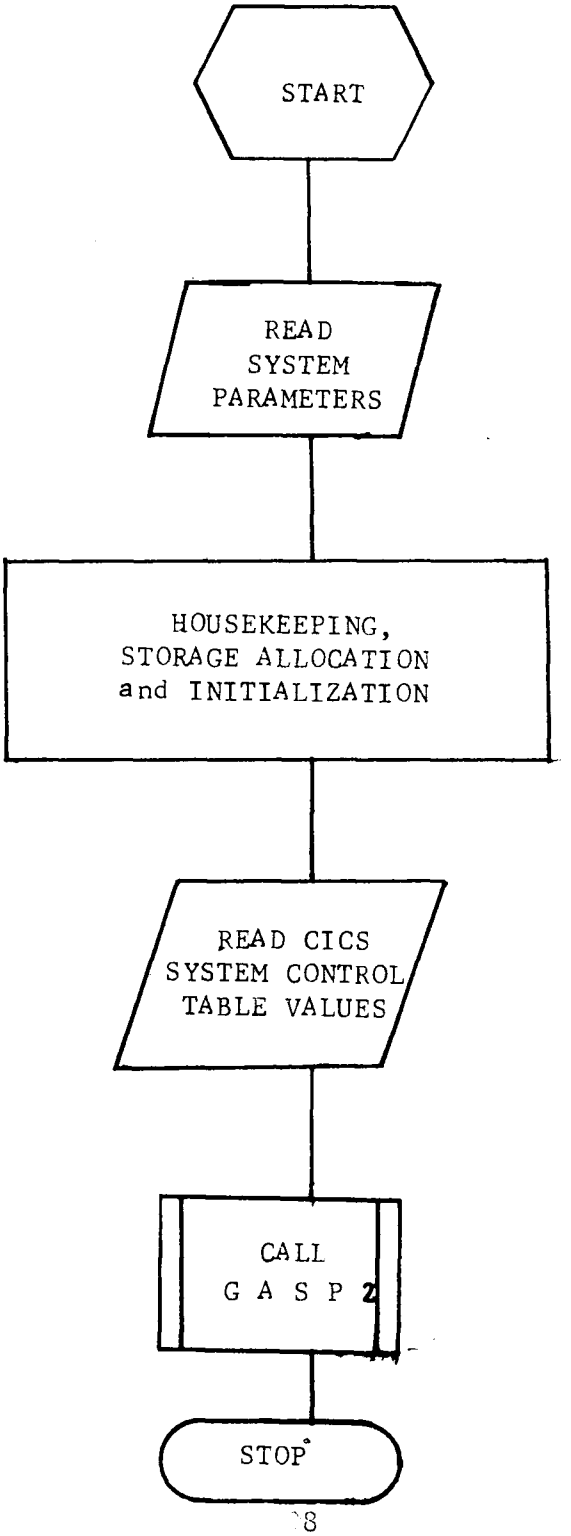
The last routine to be called in a simulation run is subroutine END_SIM. This routine is used to close out all time-generated statistics used in the model. This ensures that all statistics are updated to their final status at the end of the run. Finally, END_SIM sets variable MSTOP to -1 to end the simulation and variable NORPT to zero to request the final summary reports.

# APPENDIX E

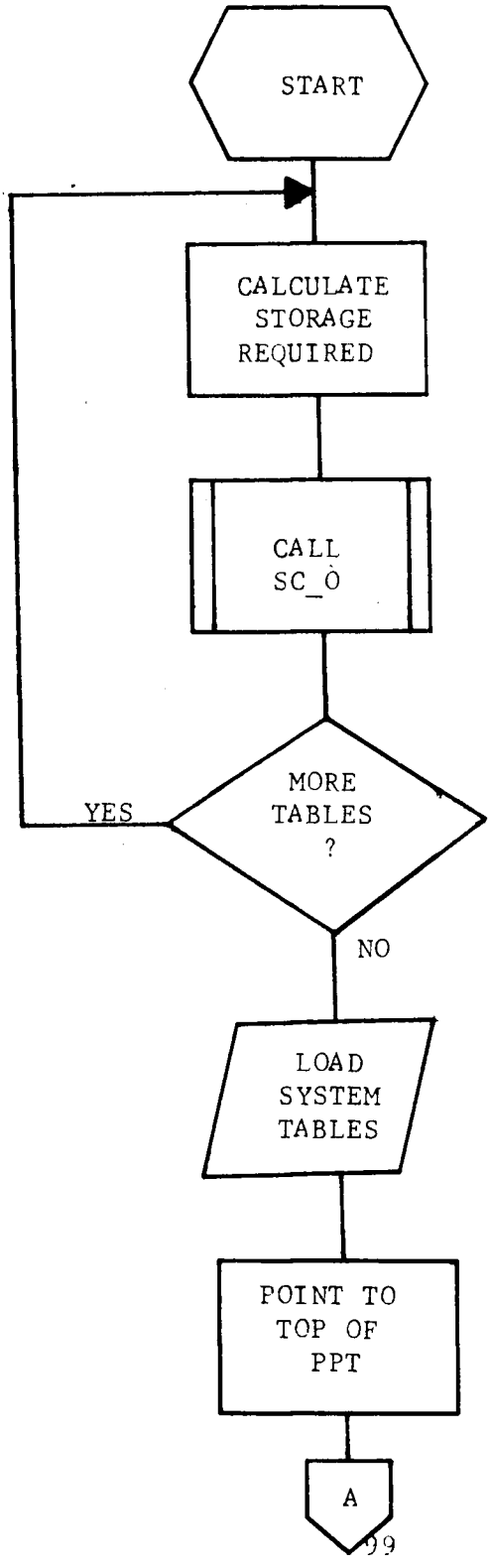
The following set of program flowcharts represent the logic flow of all user routines within the model. No attempt was made to represent each program statement in these flowcharts. It was considered more important to represent the flow of processes through the routines.

MAIN

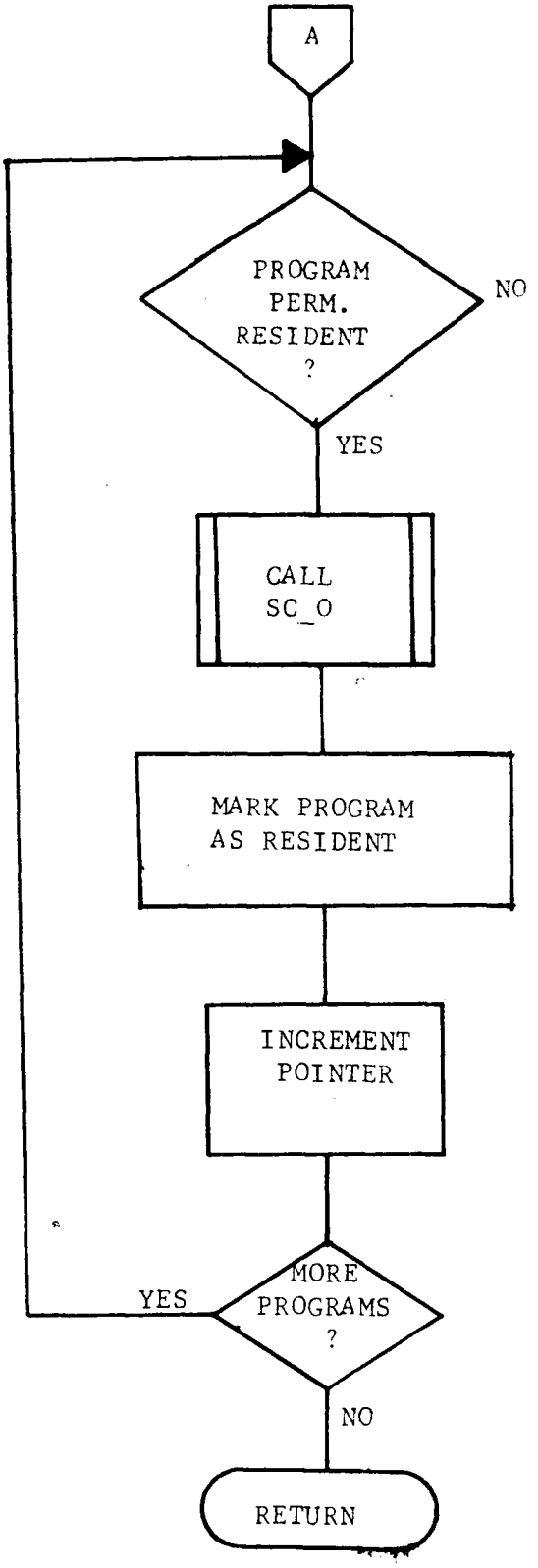
FIGURE 2



SYSINIT FIGURE 3

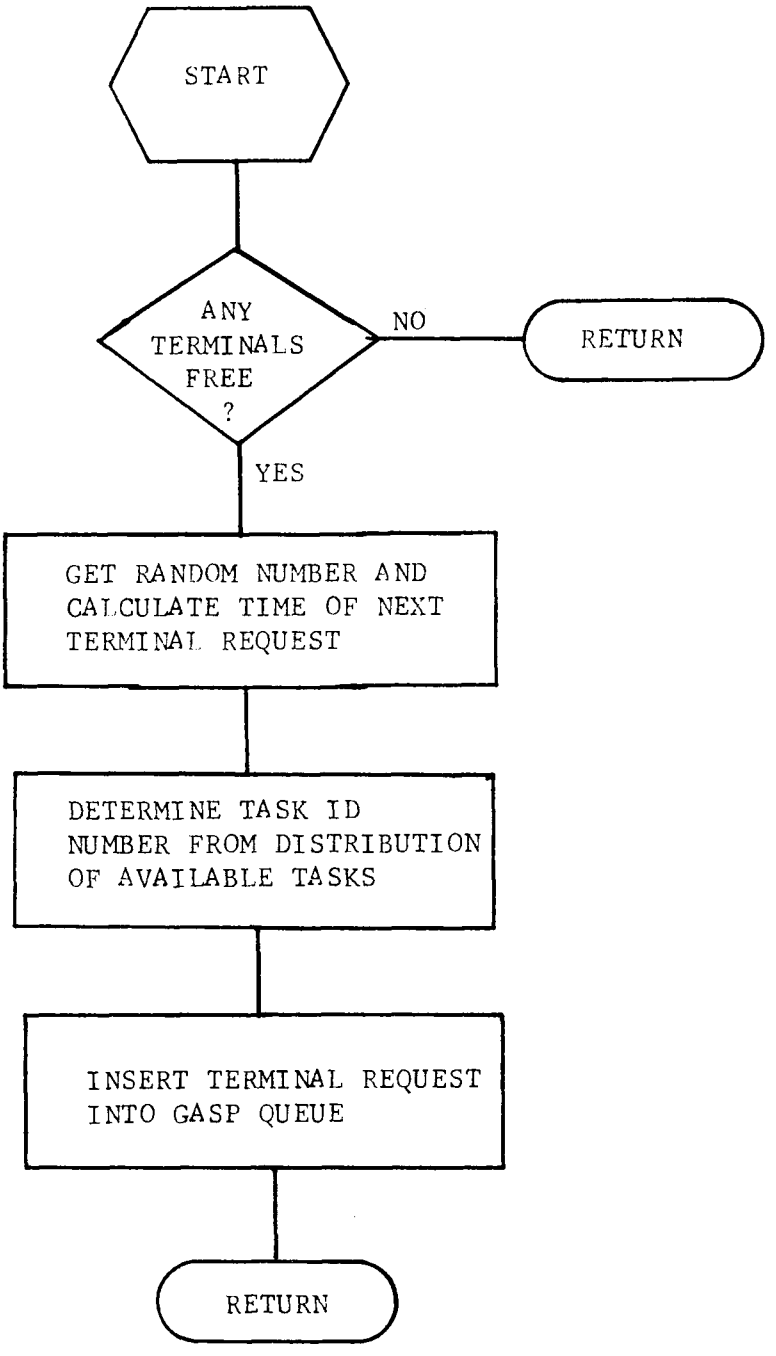



Allocate storage for all CICS System Control Tables

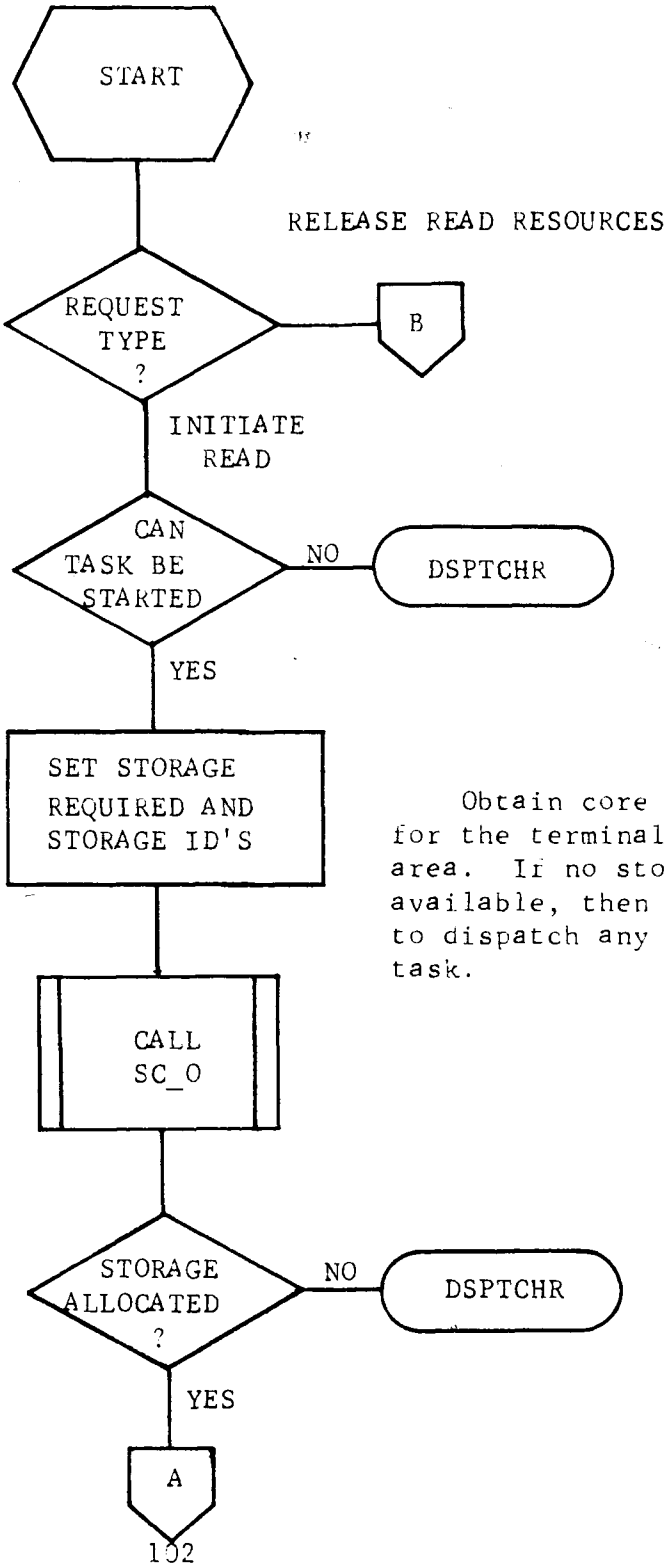



Obtain storage for all programs which are marked as being 'permanently resident' in the Processing Program Table (PPT)

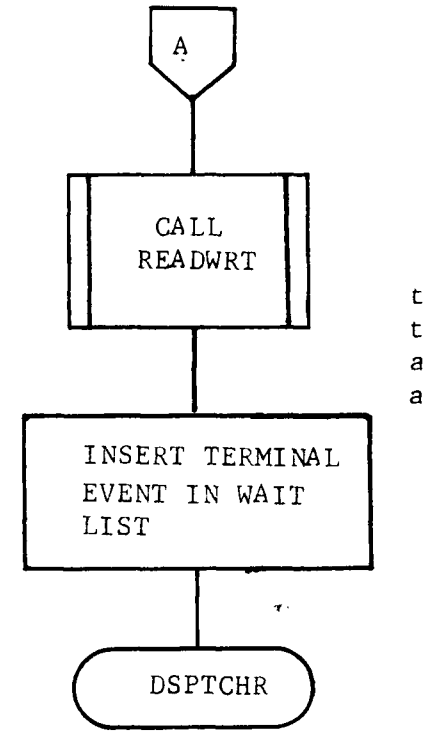
TC_NEXT FIGURE 4



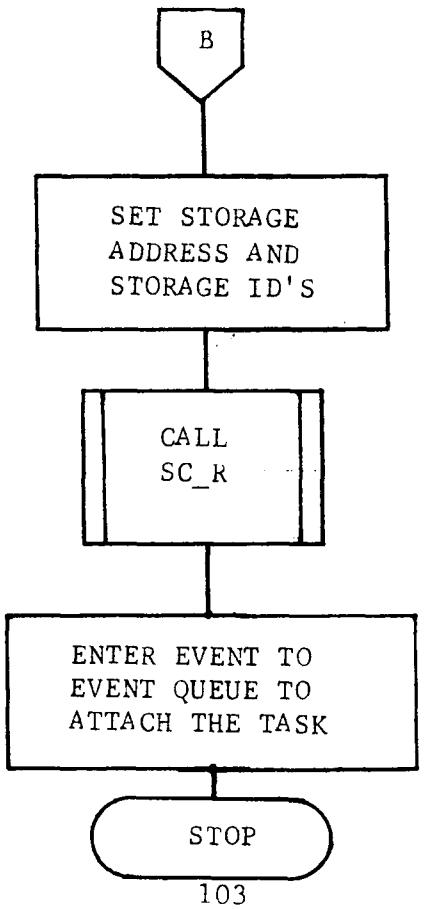
TC_GET FIGURE 5




Obtain core storage for the terminal input area. If no storage is available, then return to dispatch any existing task.

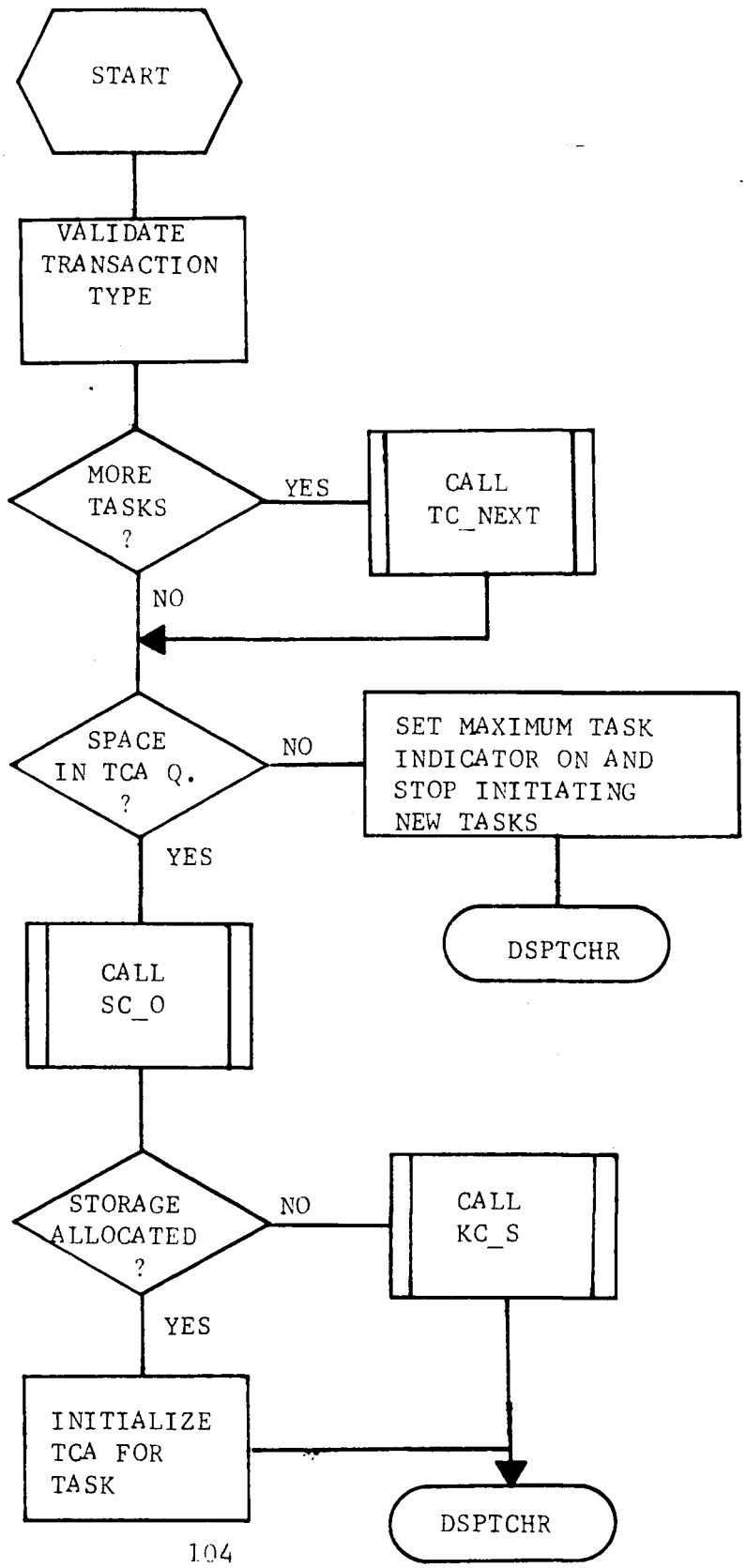



Initiate the read from the terminal, insert the task into the wait list and return to dispatch any existing task.

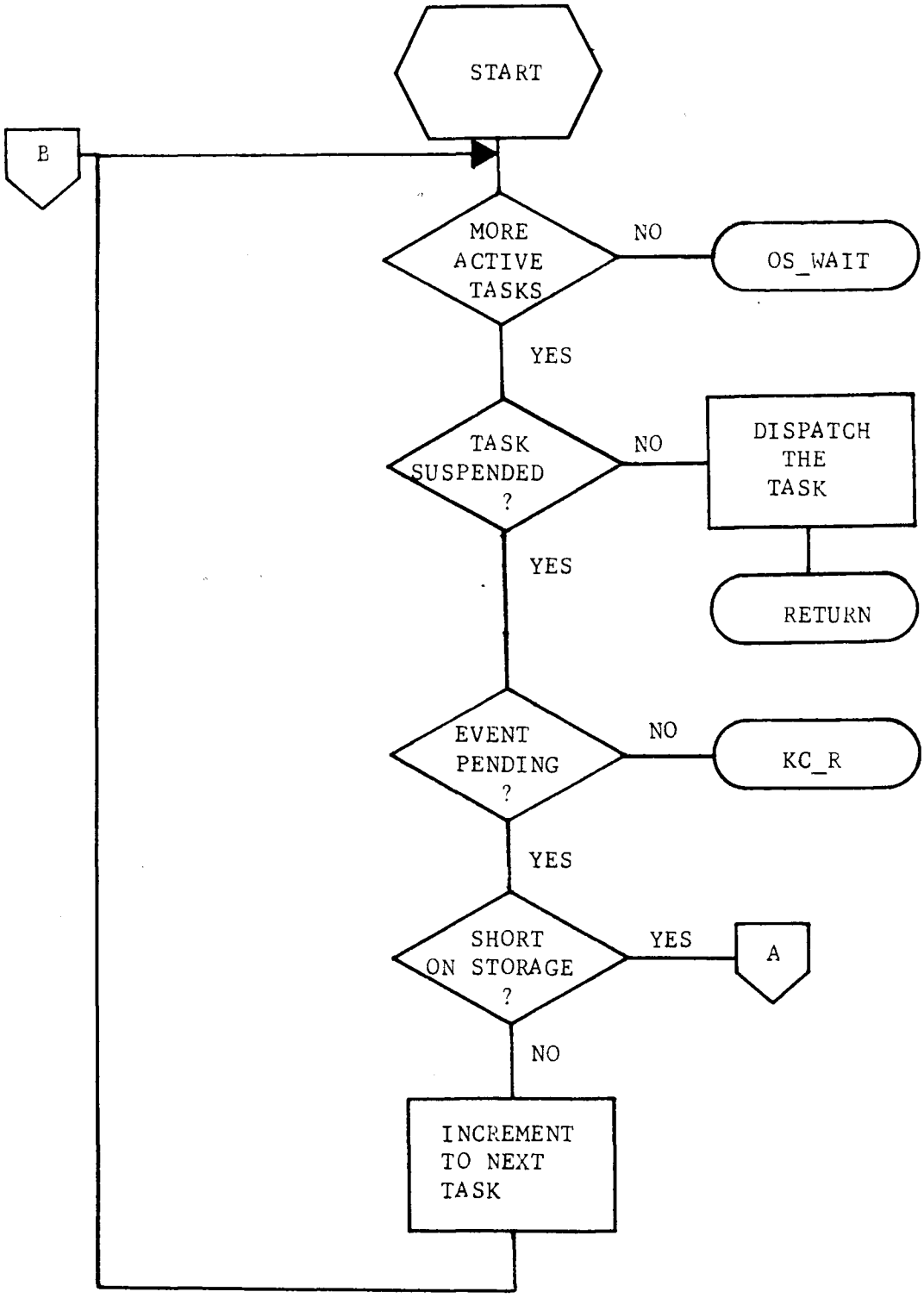


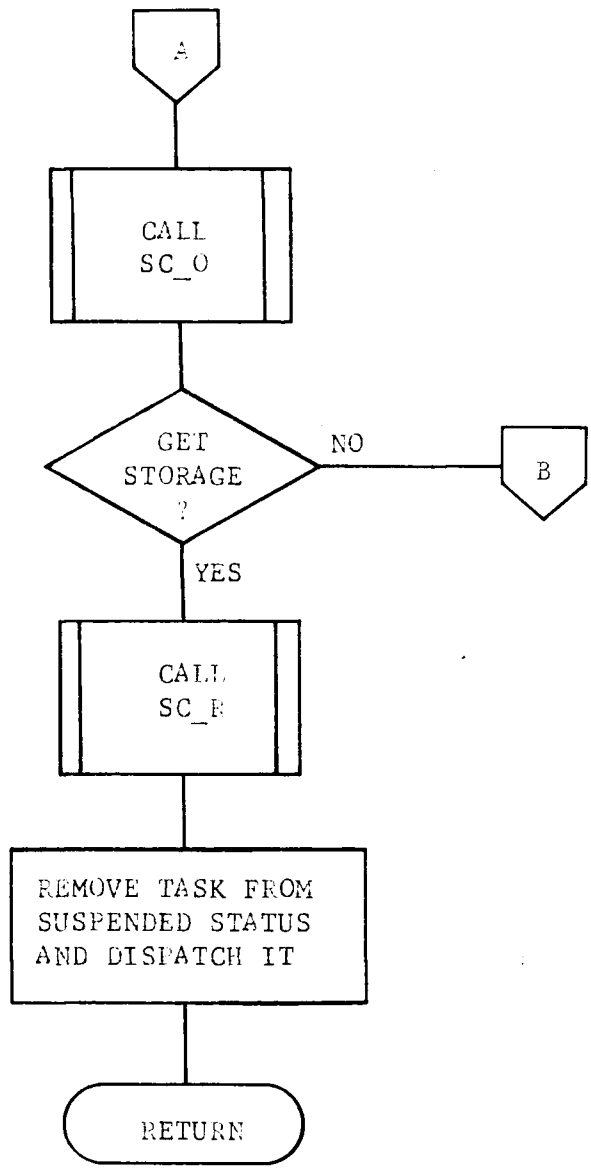

Release storage acquired for terminal input.

KC_A FIGURE 6

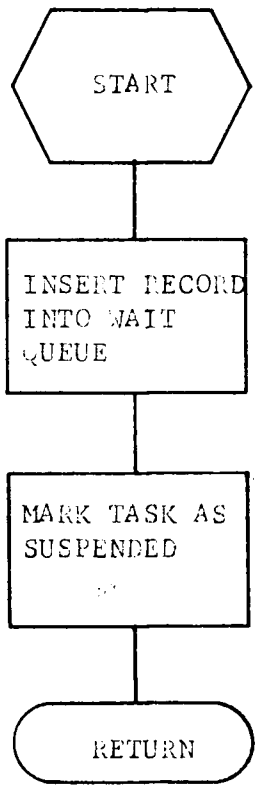

START
VALIDATE TRANSACTION TYPE
MORE TASKS ?
YES
CALL TC_NEXT
NO
SPACE IN TCA Q. ?
NO
SET MAXIMUM TASK INDICATOR ON AND STOP INITIATING NEW TASKS
DSPTCHR
YES
CALL SC_O
STORAGE ALLOCATED ?
NO
CALL KC_S
YES
INITIALIZE TCA FOR TASK
DSPTCHR

DSPTCHR FIGURE 7

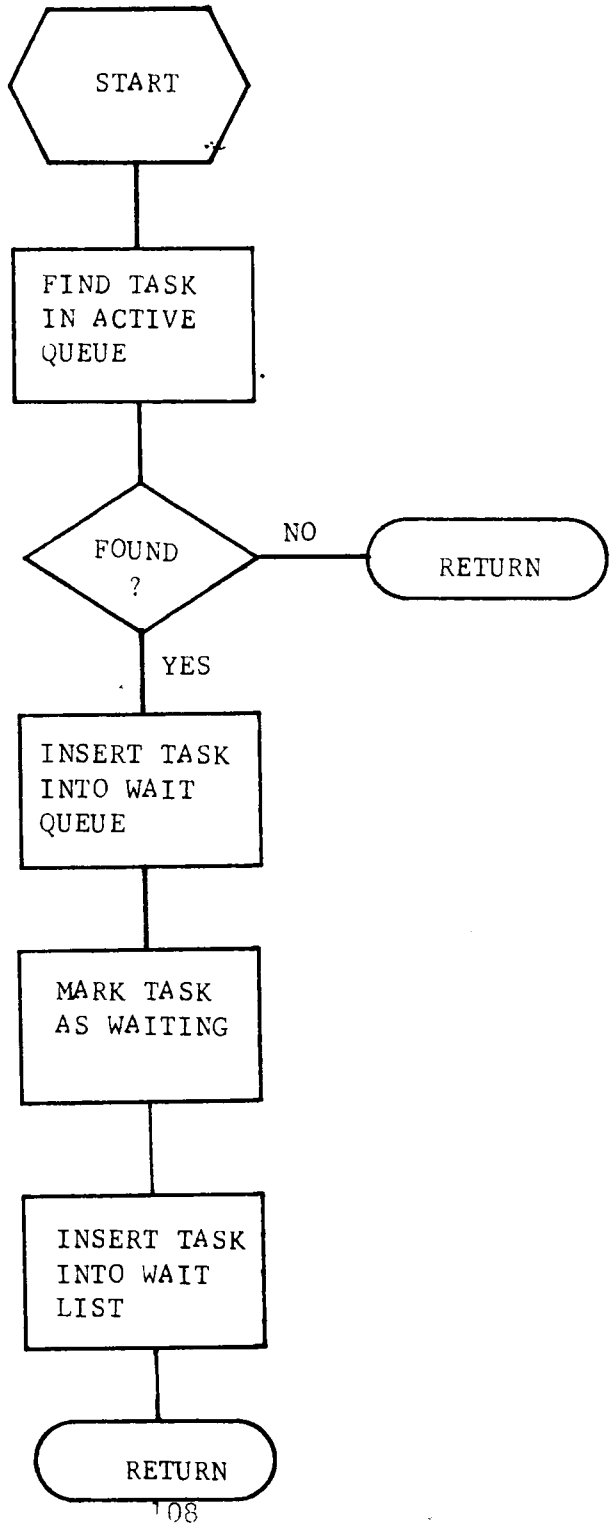



A
CALL
SC_O
GET
STORAGE
?
NO
B
YES
CALL
SC_E
REMOVE TASK FROM
SUSPENDED STATUS
AND DISPATCH IT
RETURN

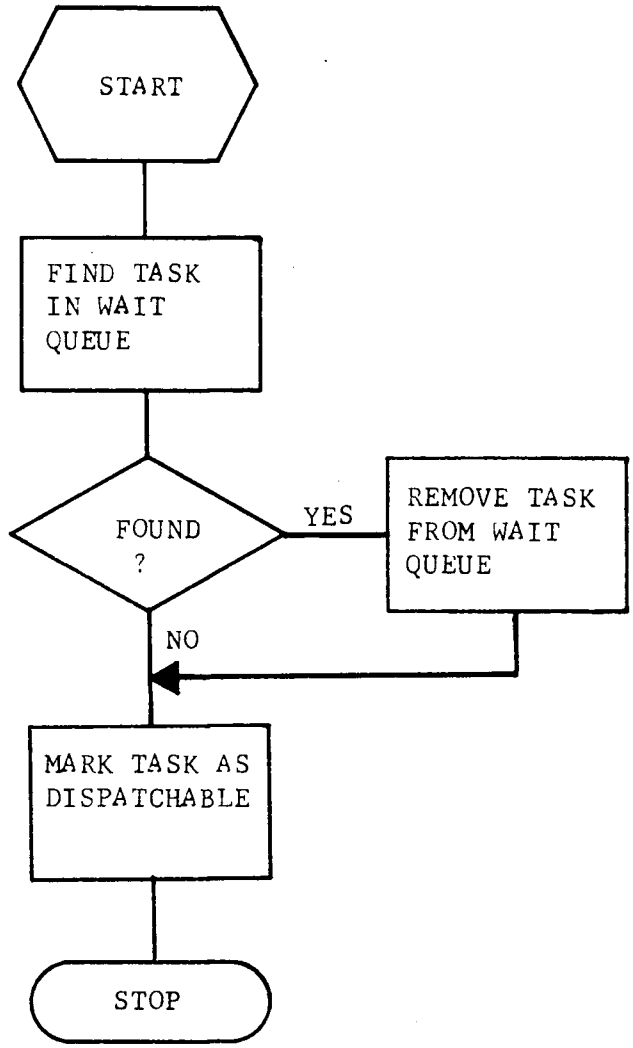
KC_S FIGURE 3


START
INSERT RECORD INTO WAIT QUEUE
MARK TASK AS SUSPENDED
RETURN

KC_W FIGURE 9

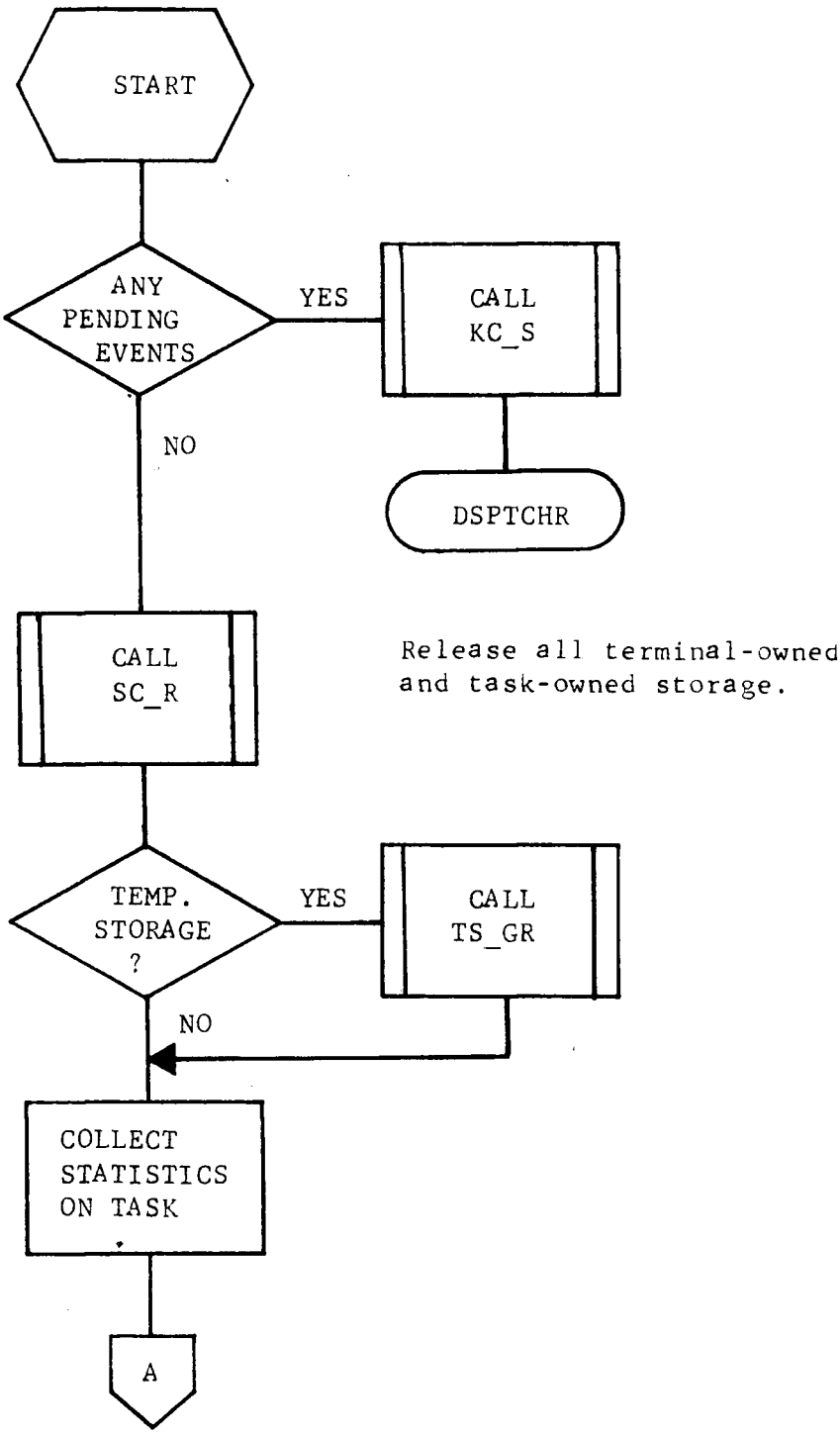

START
FIND TASK IN ACTIVE QUEUE
FOUND ?
NO
RETURN
YES
INSERT TASK INTO WAIT QUEUE
MARK TASK AS WAITING
INSERT TASK INTO WAIT LIST
RETURN

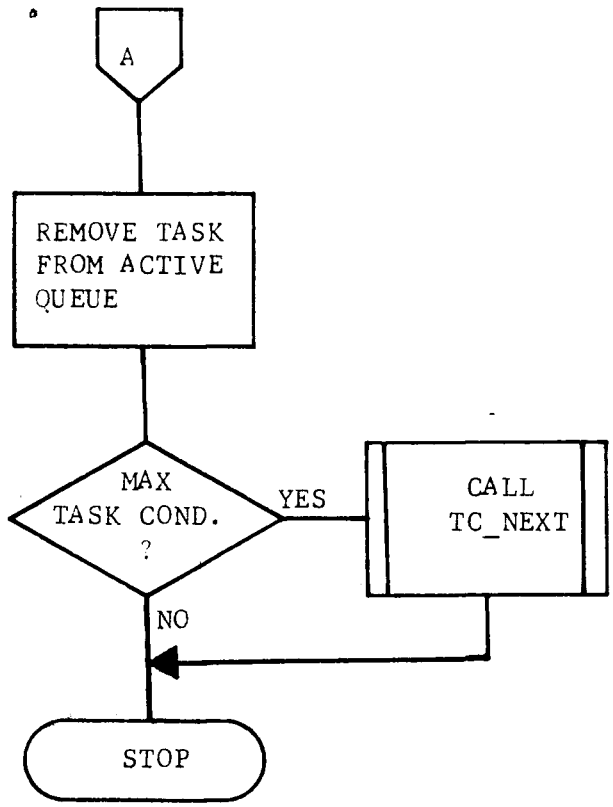
KC_R FIGURE 10



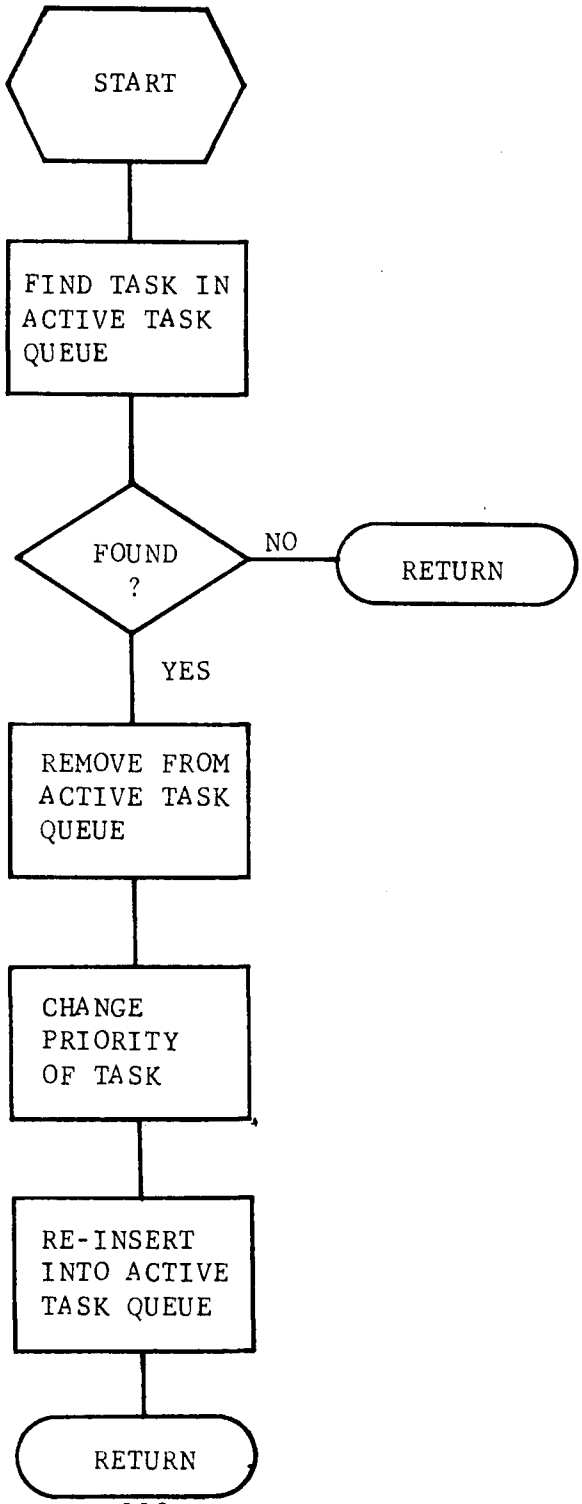
KC_T

FIGURE 11



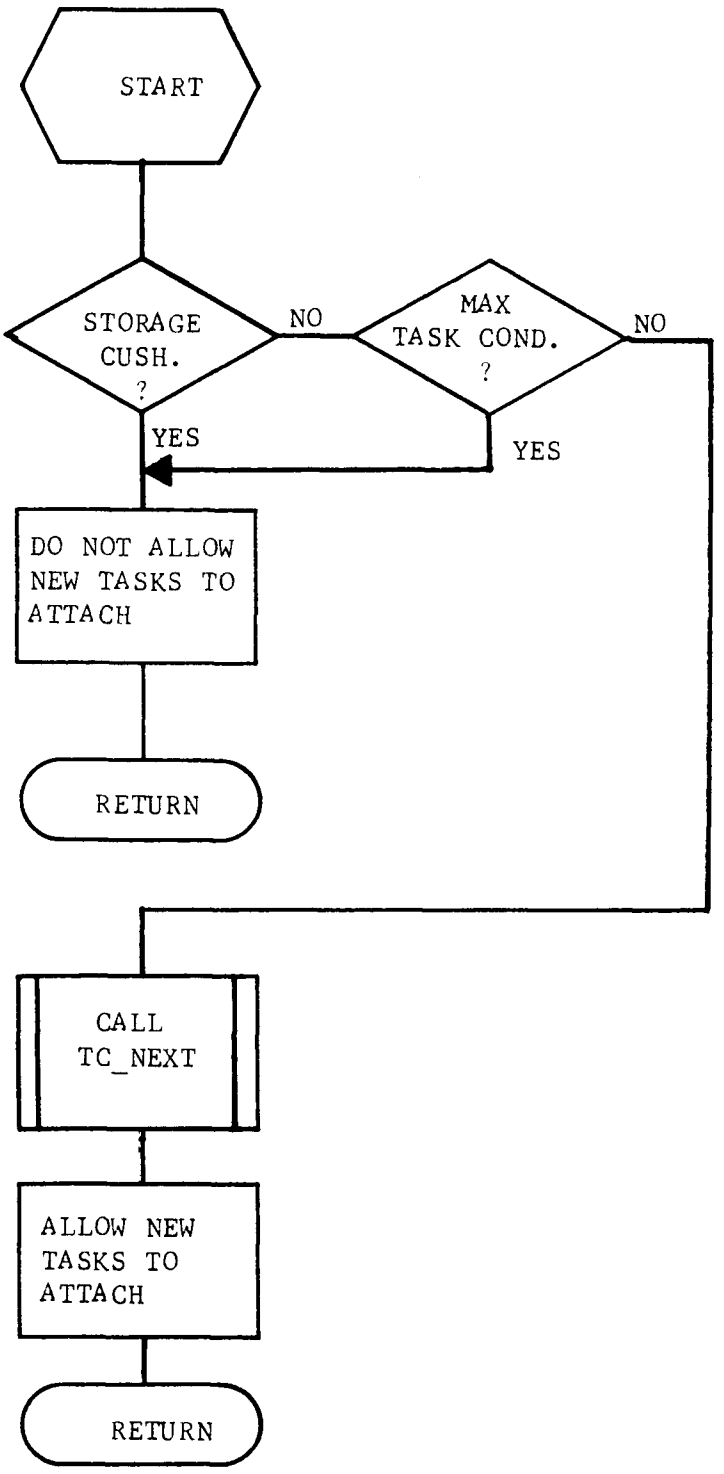

A
REMOVE TASK FROM ACTIVE QUEUE
MAX TASK COND. ?
YES
CALL TC_NEXT
NO
STOP

KC_C FIGURE 12



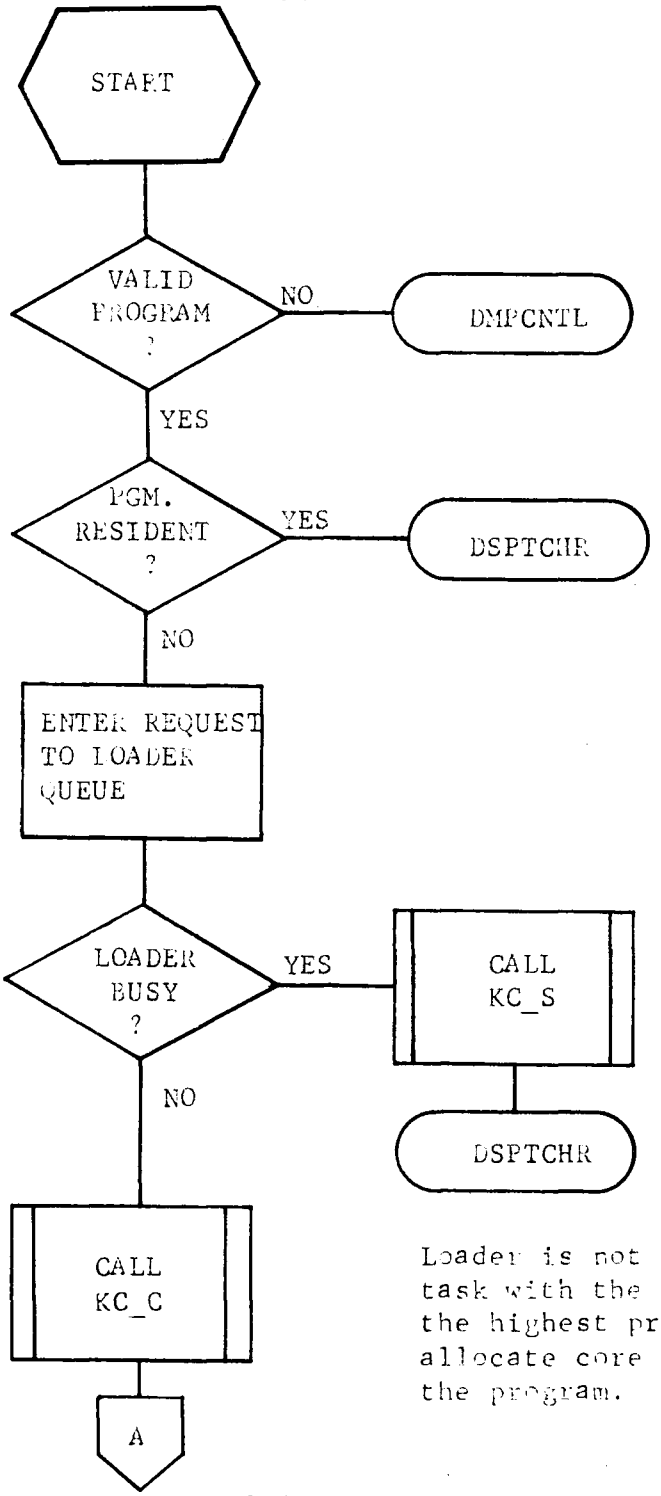
KC_RS

FIGURE 13

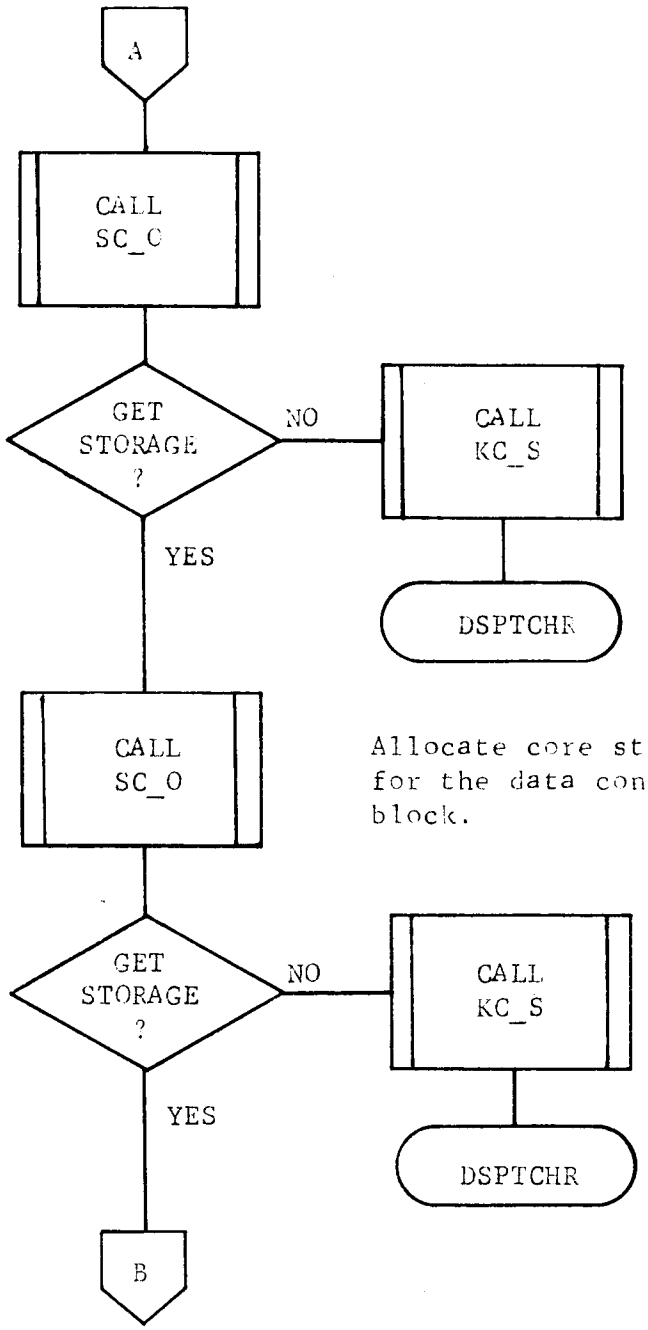


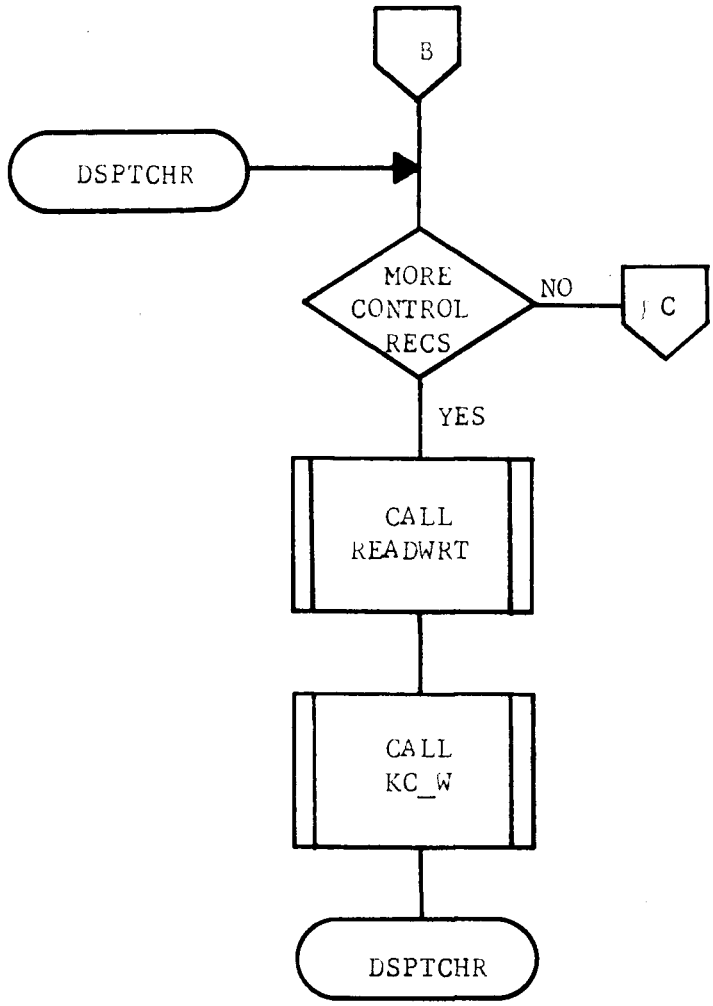
PC_F

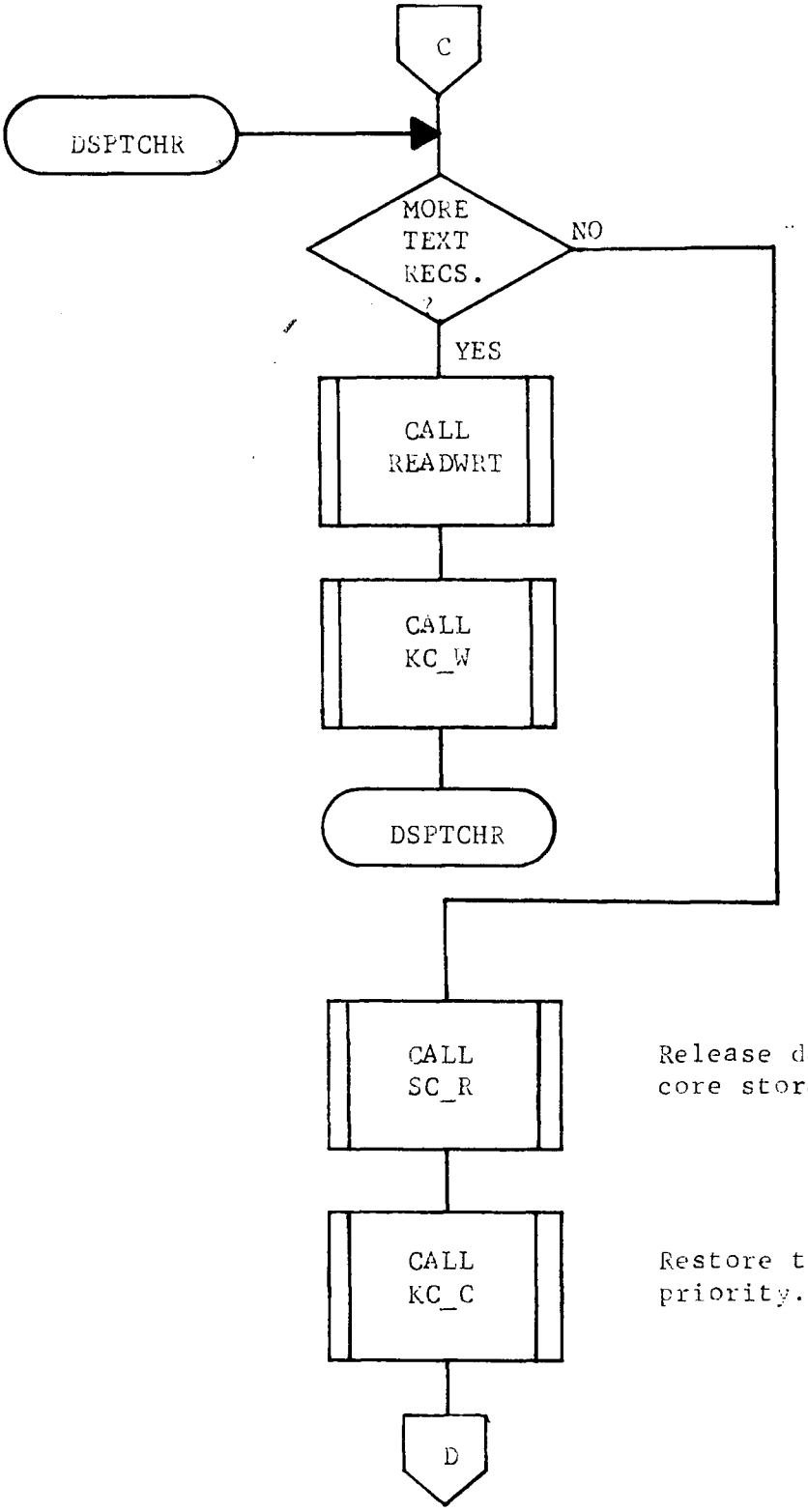
FIGURE 14

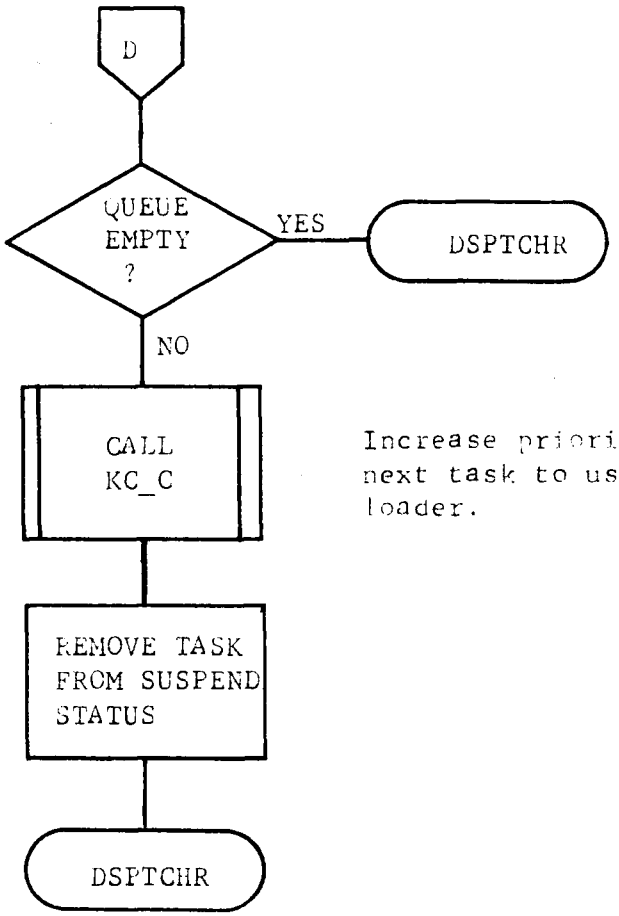



Loader is not busy. Give task with the load request the highest priority and allocate core storage for the program.


A
CALL
SC_C
GET
STORAGE
?
NO
CALL
KC_S
DSPTCHR
YES
CALL
SC_O
Allocate core storage for the data control block.
GET
STORAGE
?
NO
CALL
KC_S
DSPTCHR
YES
B

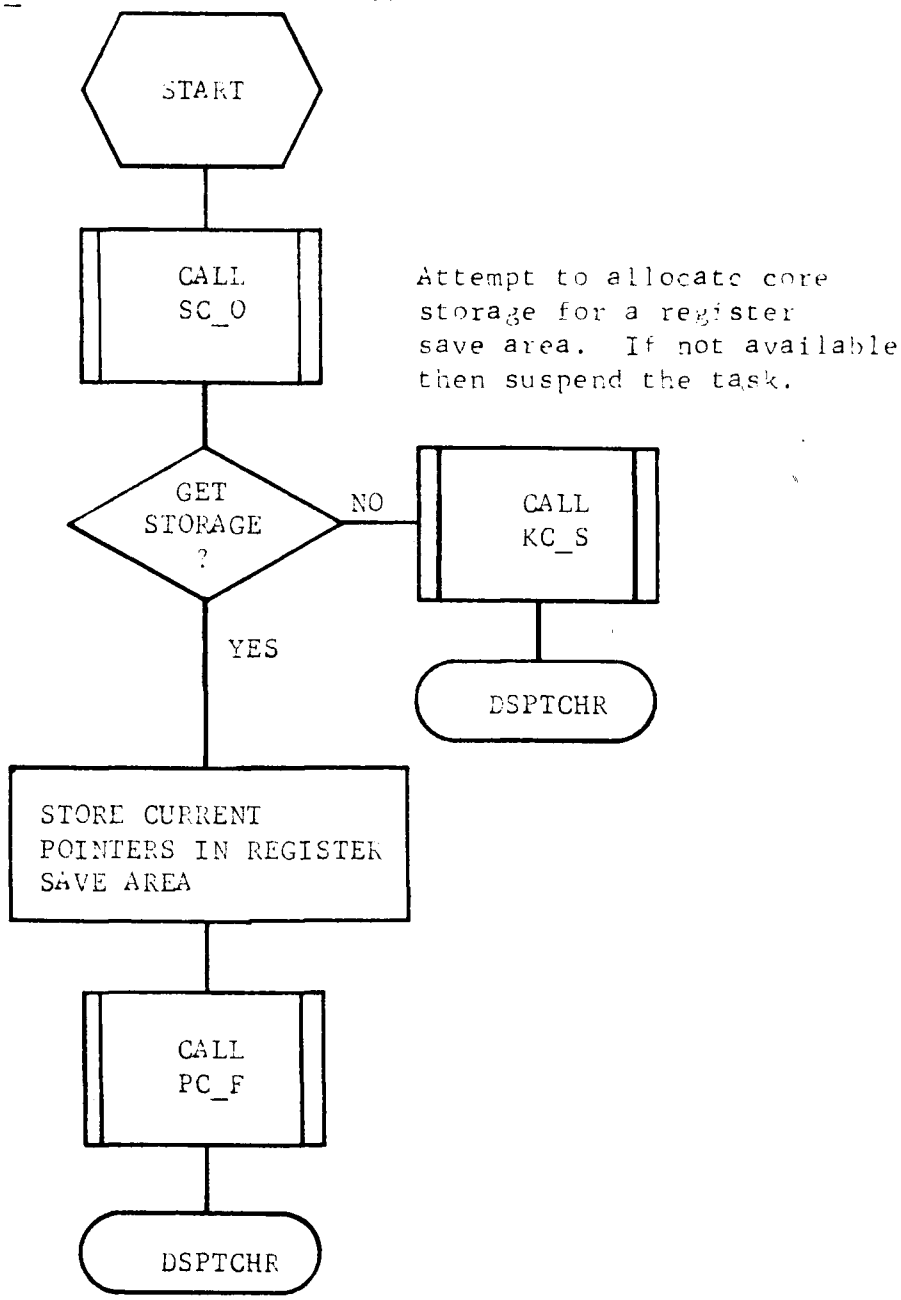

B
DSPTCHR
MORE CONTROL RECS
NO
C
YES
CALL READWRT
CALL KC_W
DSPTCHR


C
DSPTCHR
MORE TEXT RECS.
NO
YES
CALL READWRT
CALL KC_W
DSPTCHR
CALL SC_R
Release data control block core storage.
CALL KC_C
Restore task's original priority.
D

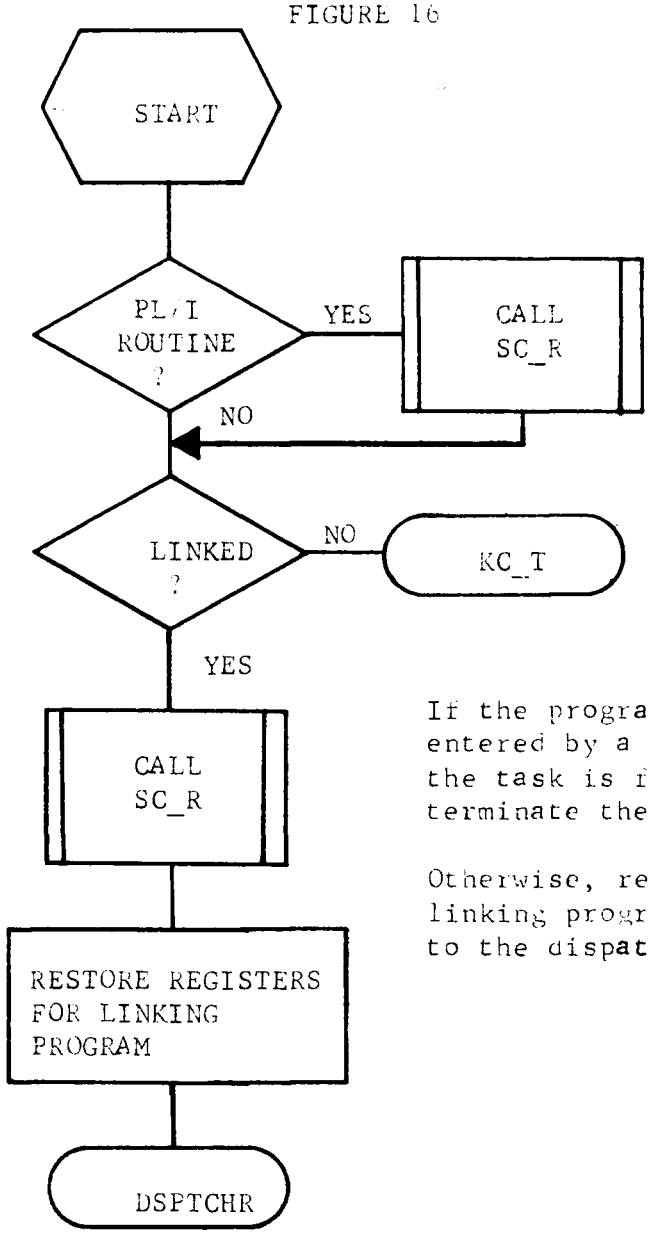



Increase priority of the next task to use the loader.

PC_L FIGURE 1b



PC_R FIGURE 16

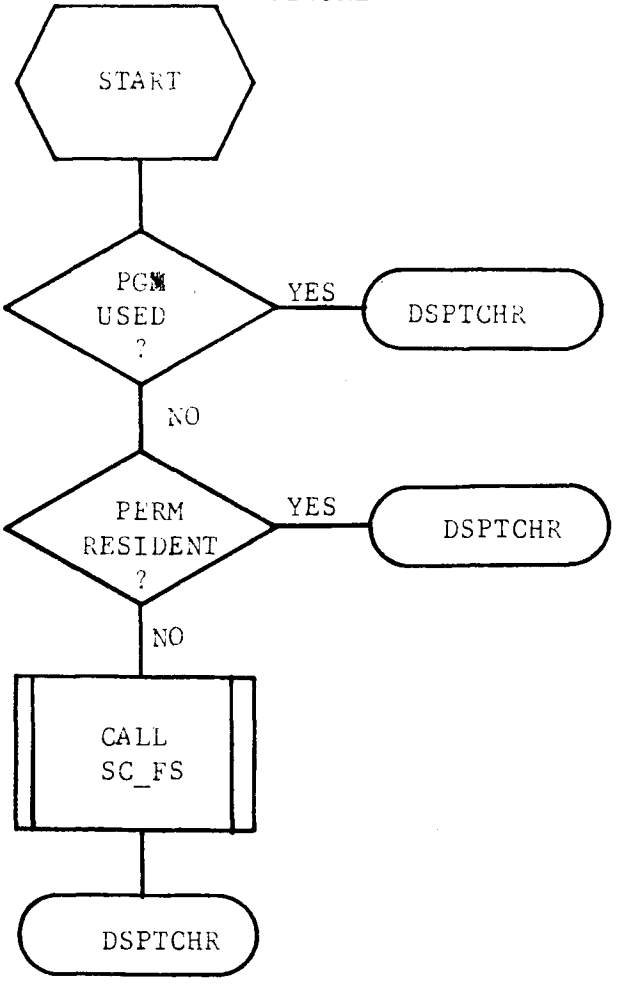



If the program was not entered by a link, then the task is finished, and terminate the task.

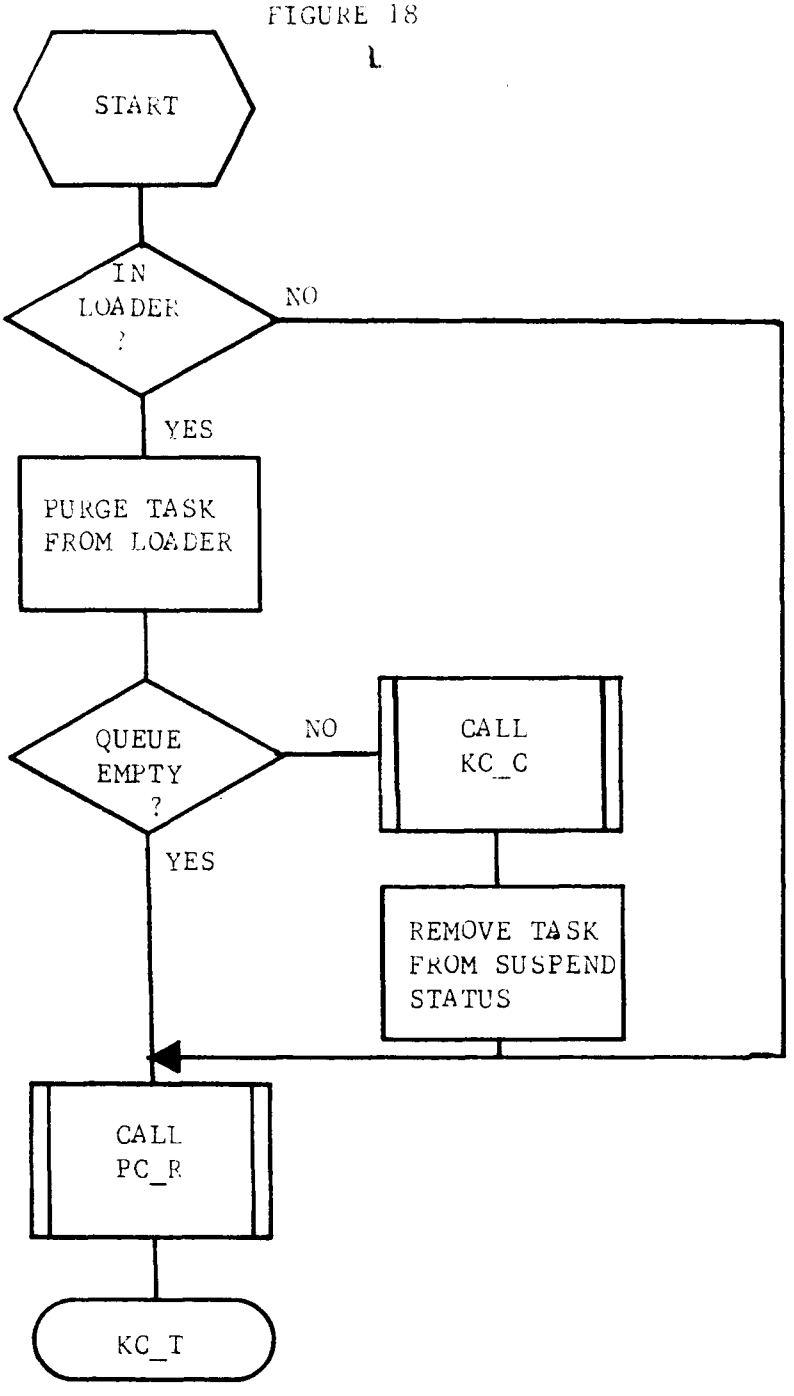
Otherwise, restore the linking program and exit to the dispatcher.

PC_D

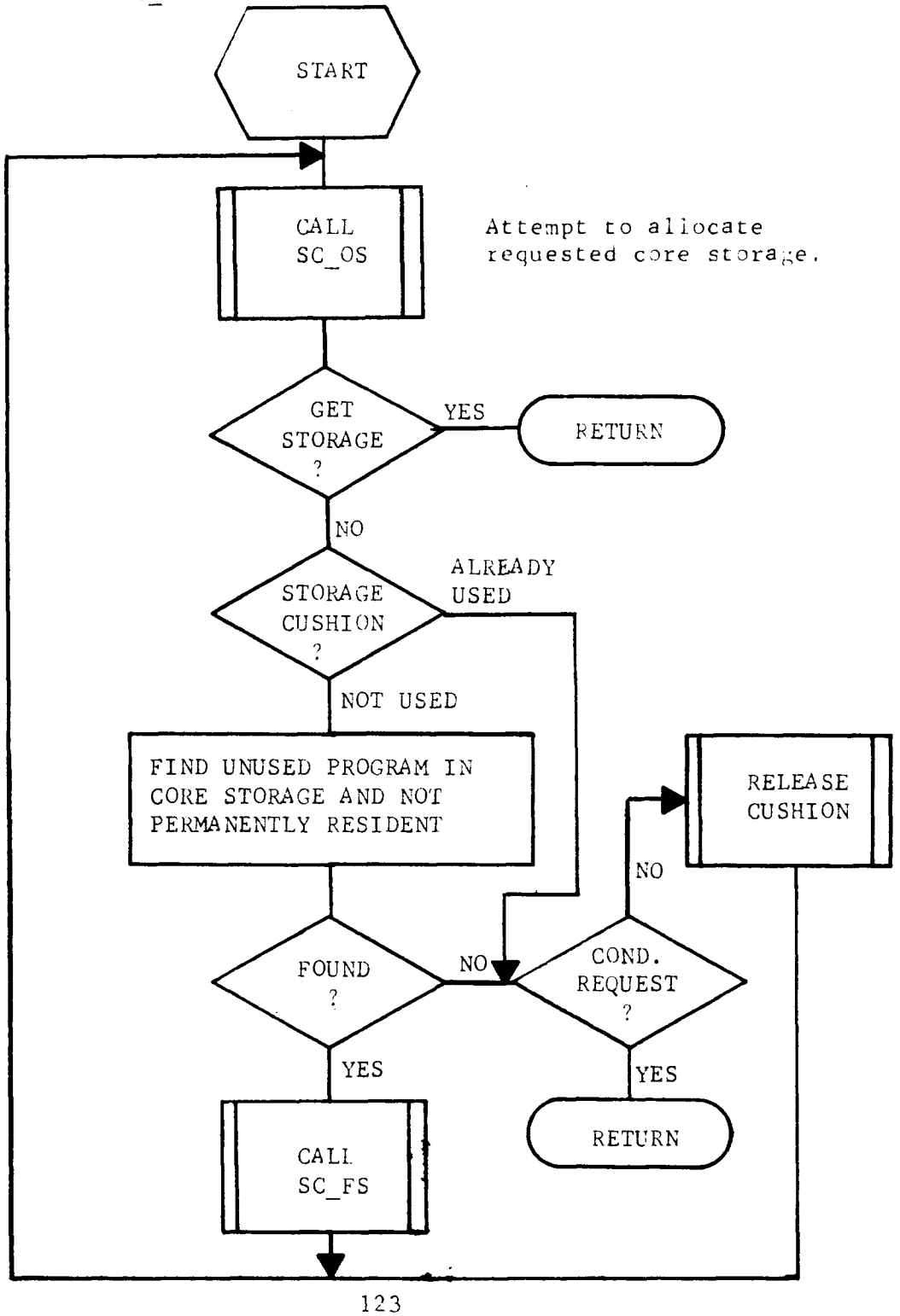
FIGURE 17



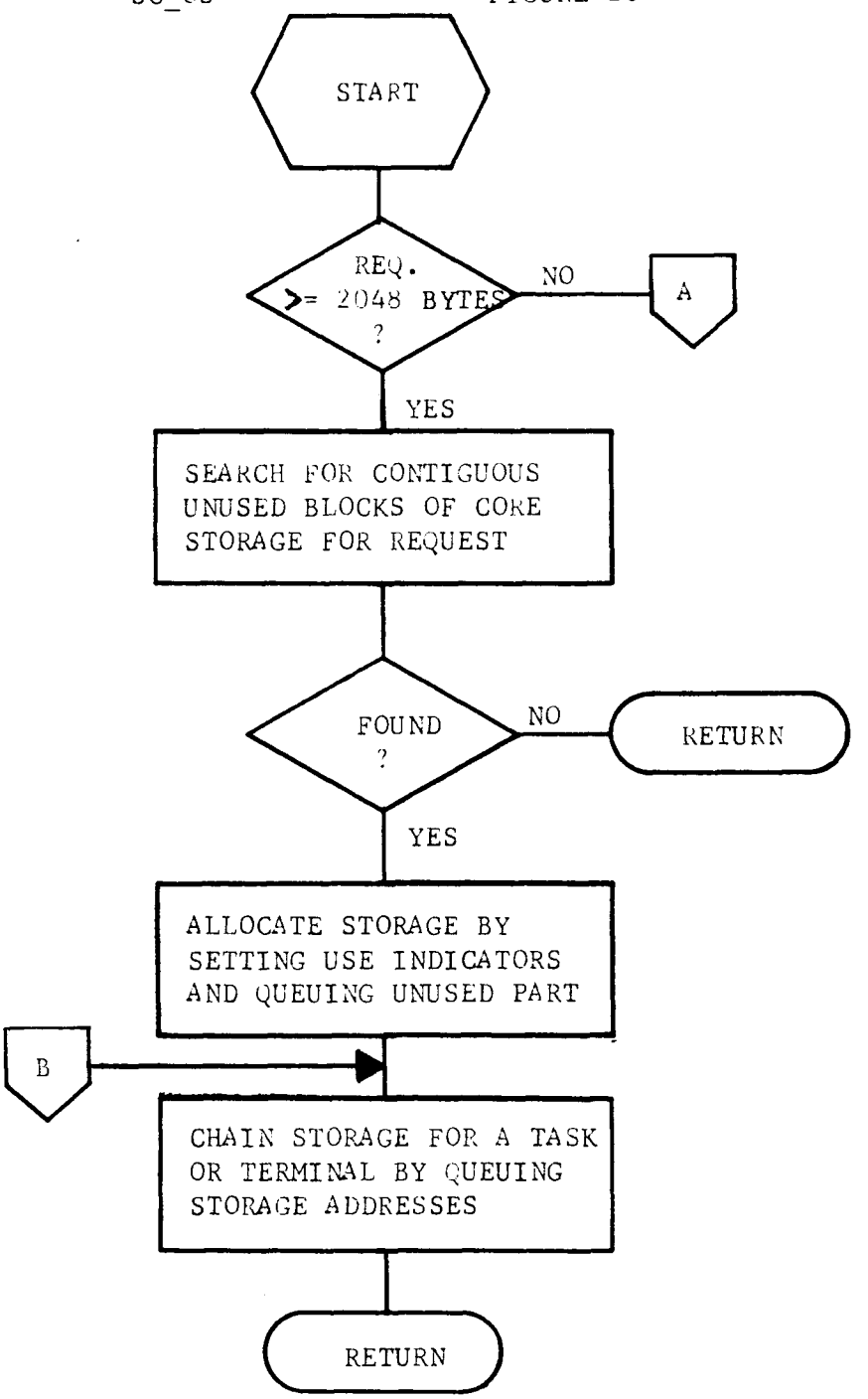
FCABEND FIGURE 18

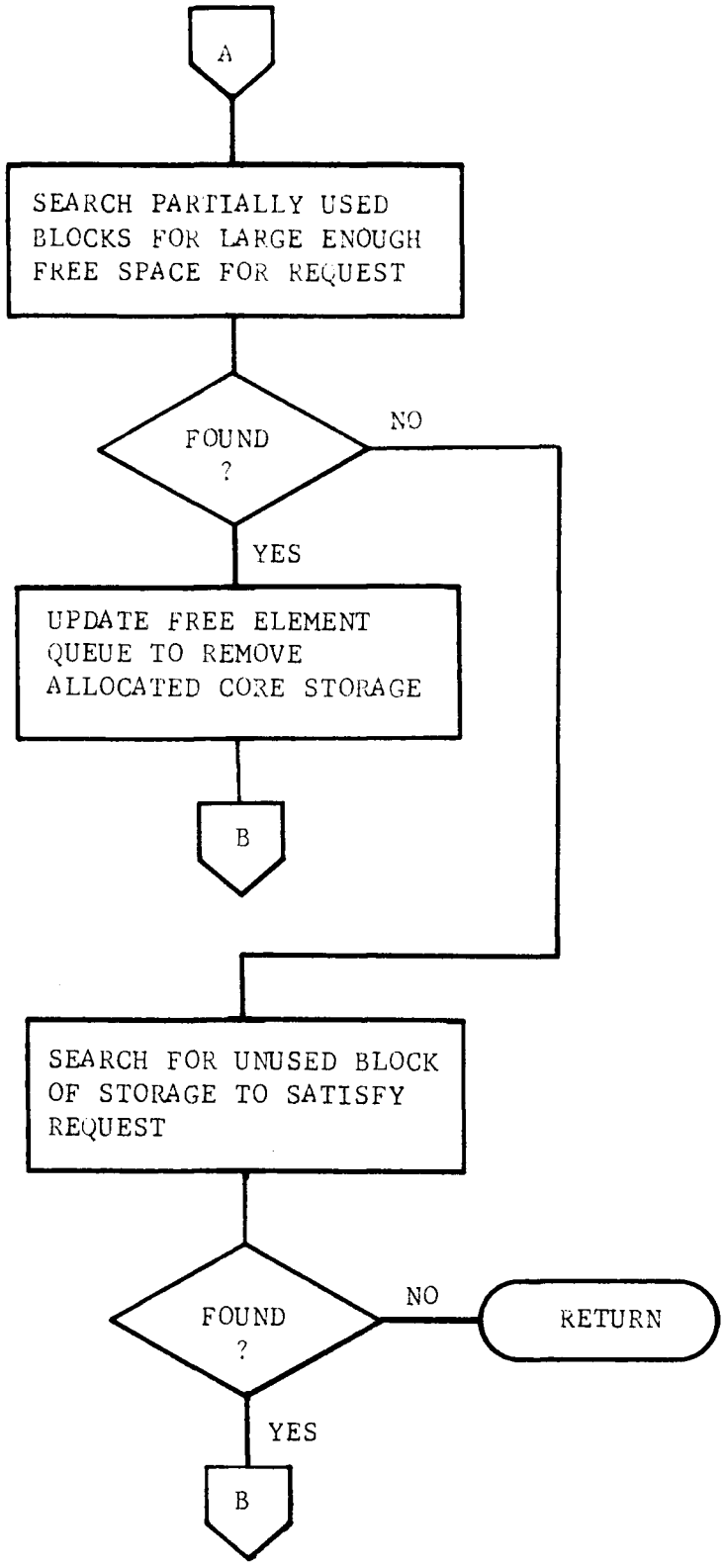


SC_O FIGURE 19

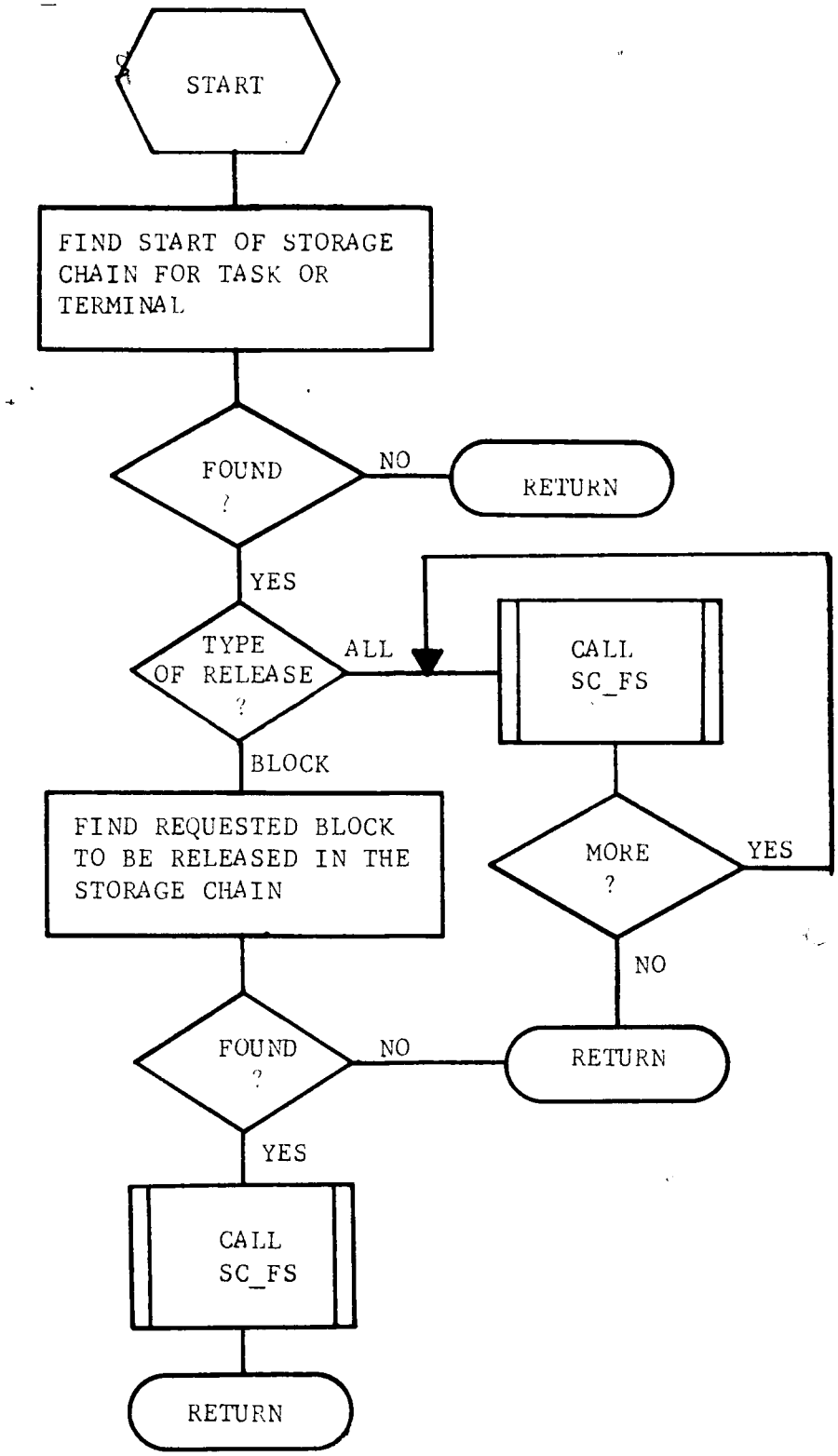


SC_OS FIGURE 20

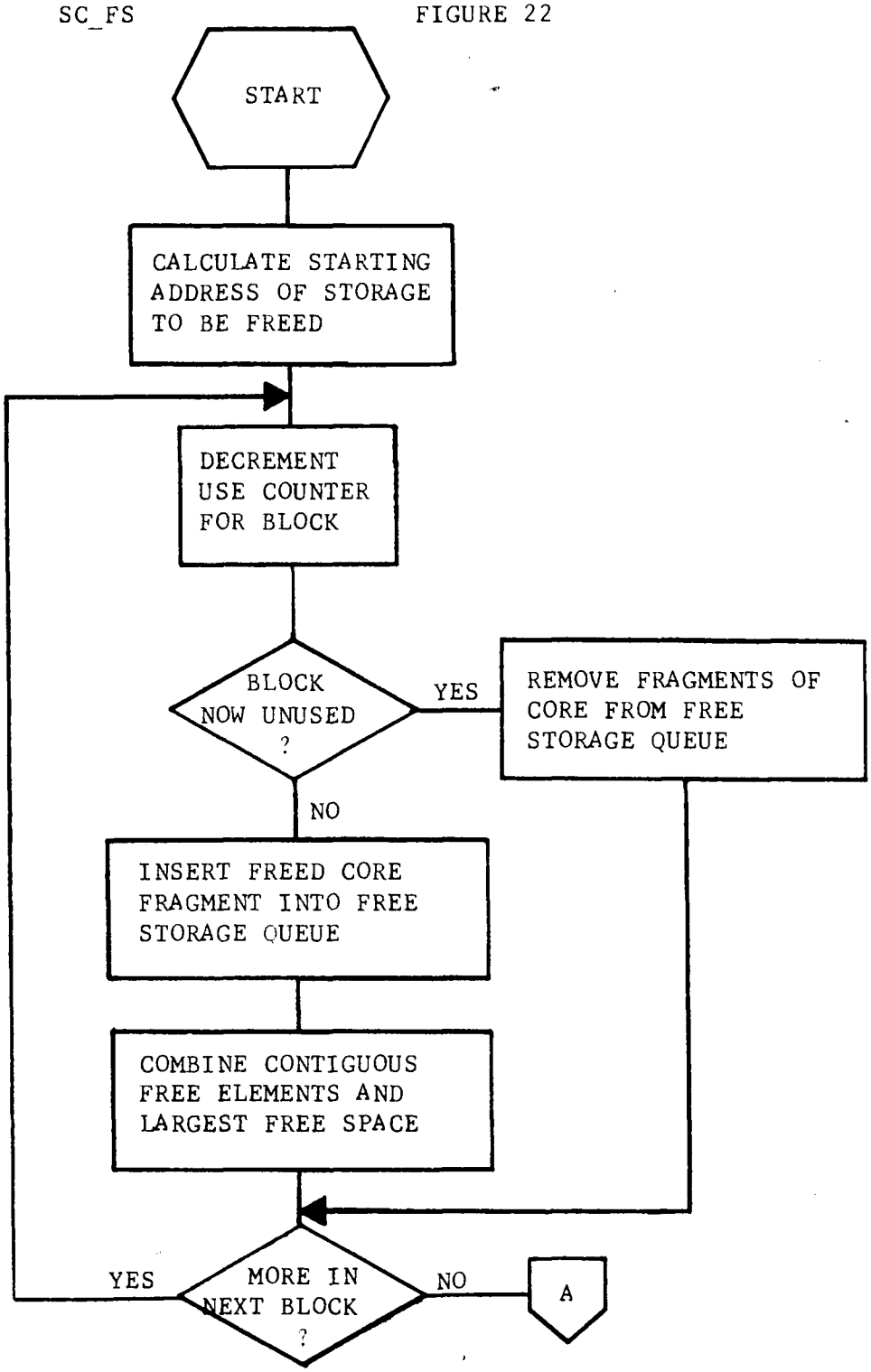


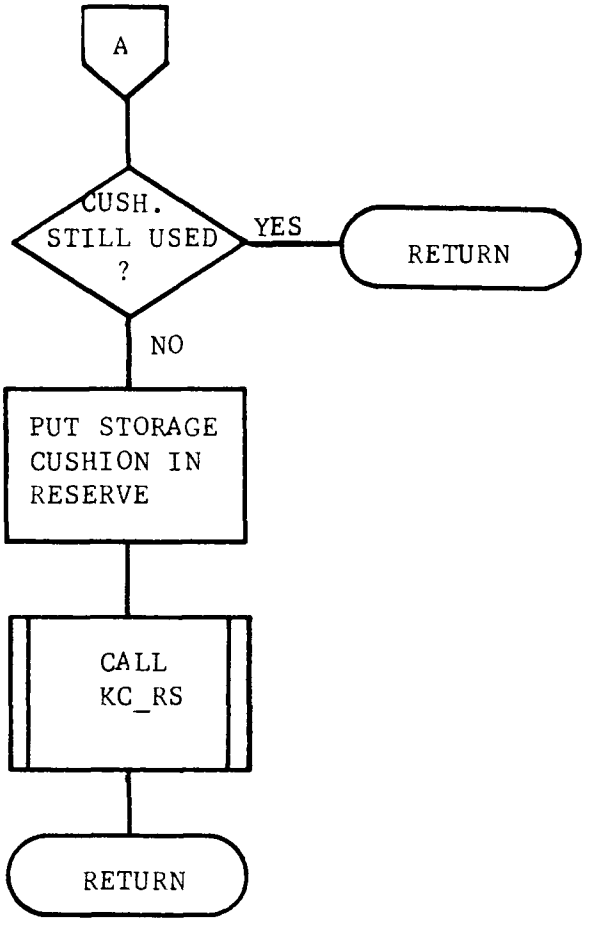

A
SEARCH PARTIALLY USED BLOCKS FOR LARGE ENOUGH FREE SPACE FOR REQUEST
FOUND ?
NO
YES
UPDATE FREE ELEMENT QUEUE TO REMOVE ALLOCATED CORE STORAGE
B
SEARCH FOR UNUSED BLOCK OF STORAGE TO SATISFY REQUEST
FOUND ?
NO
RETURN
YES
B

SC_R FIGURE 21

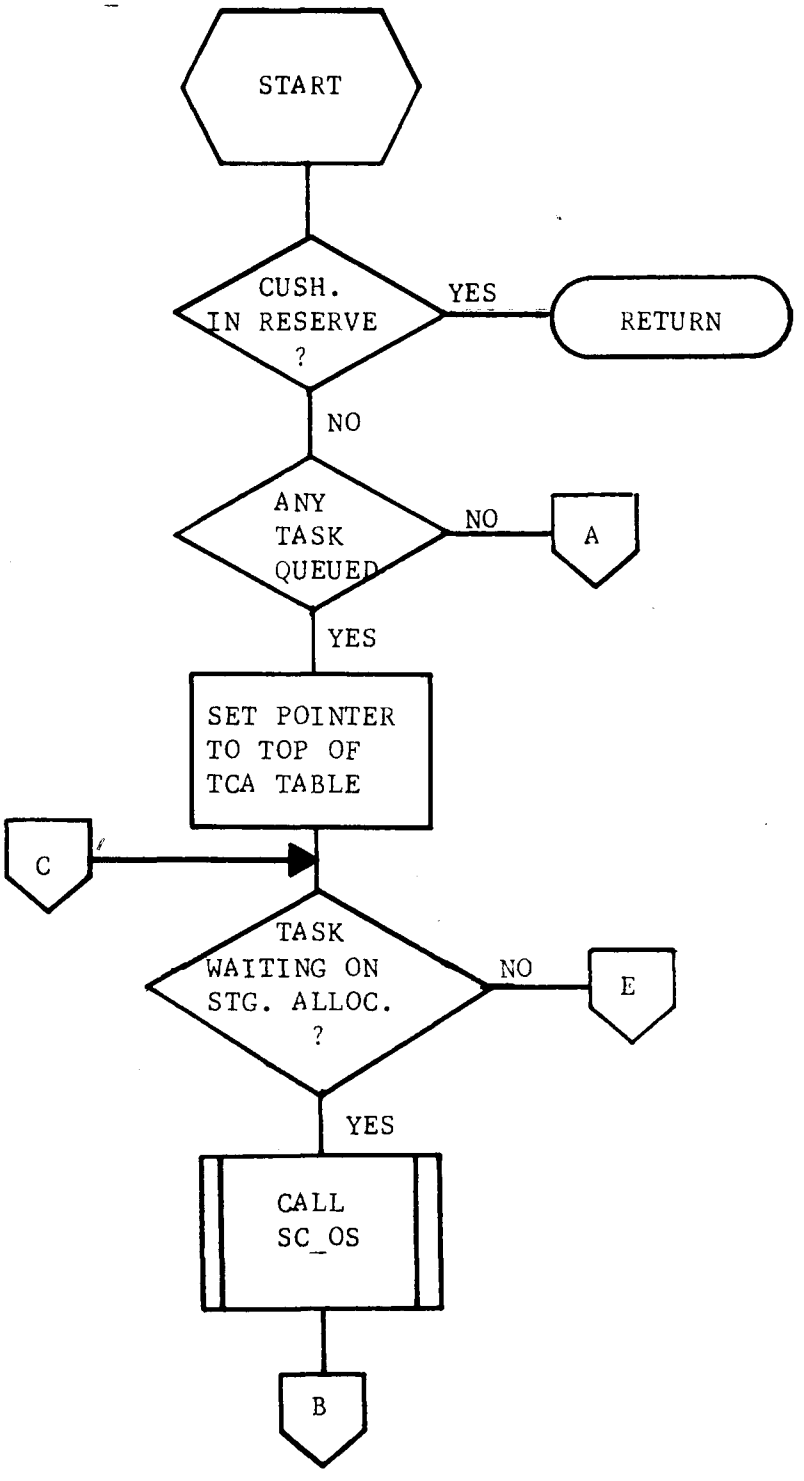


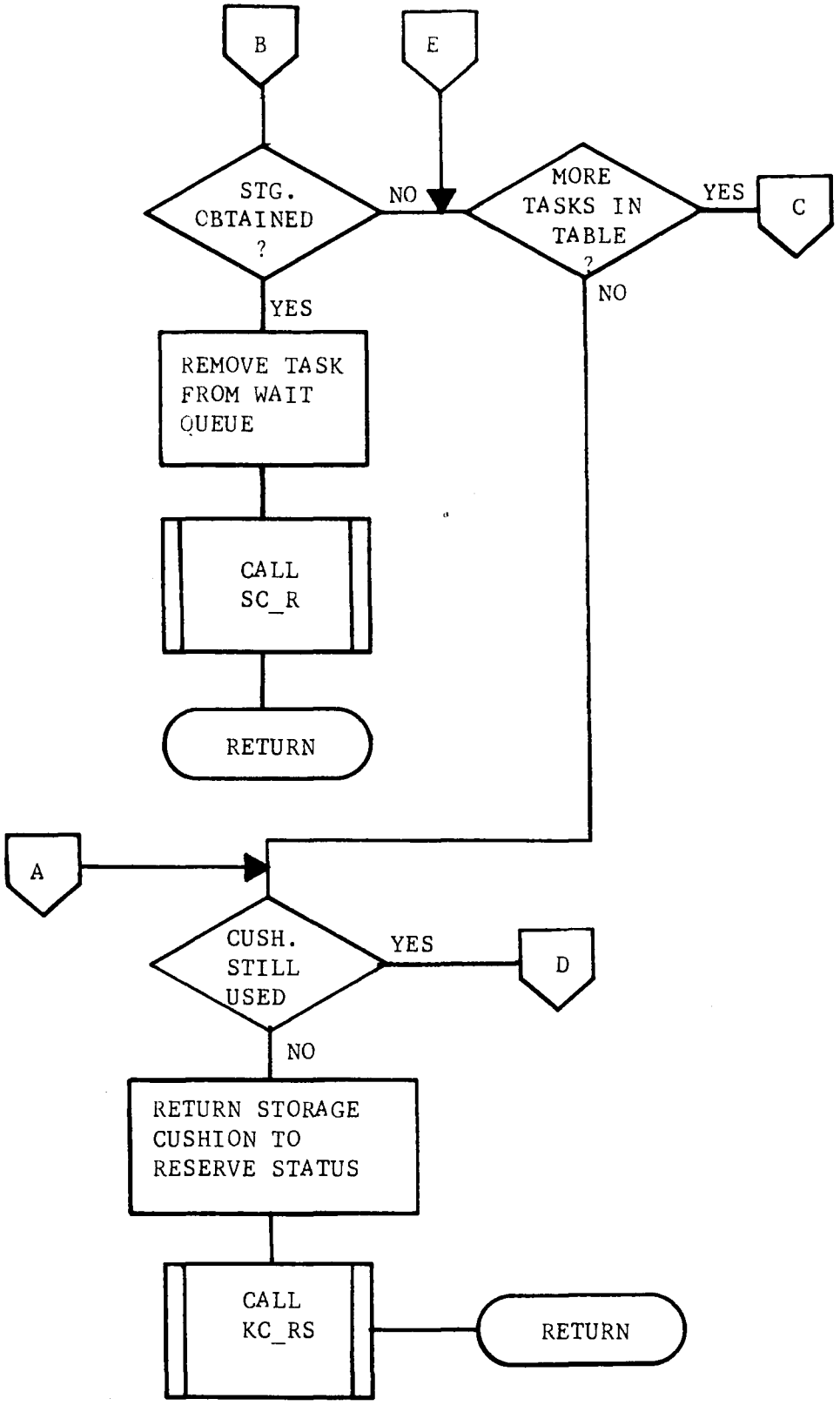
SC_FS FIGURE 22

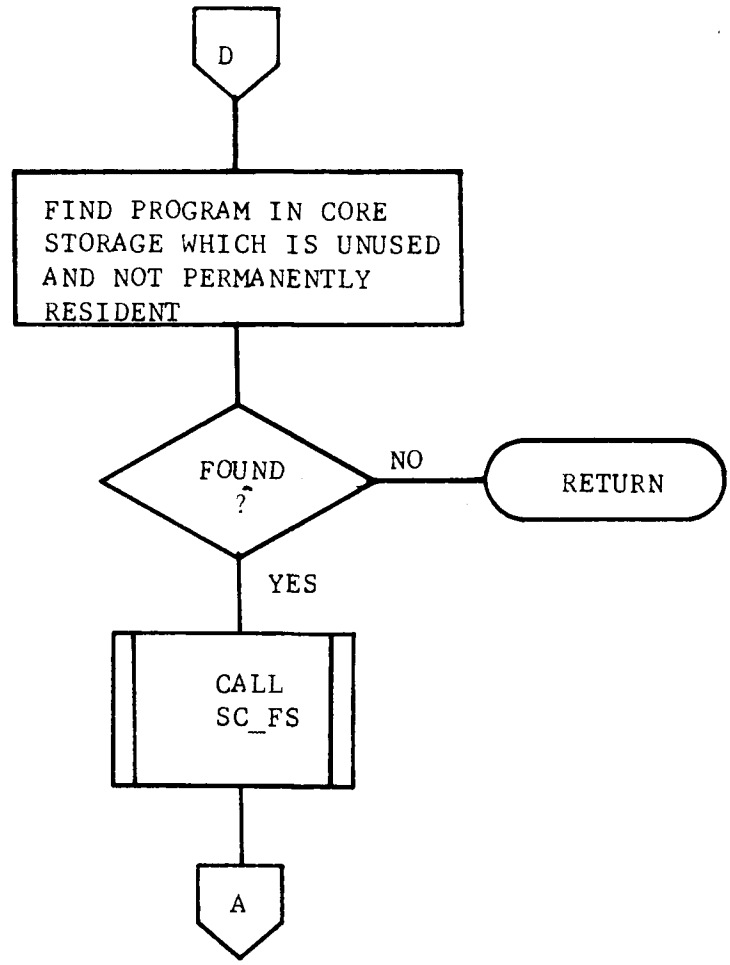



A
CUSH.
STILL USED
?
YES
RETURN
NO
PUT STORAGE
CUSHION IN
RESERVE
CALL
KC_RS
RETURN

SC_F FIGURE 23

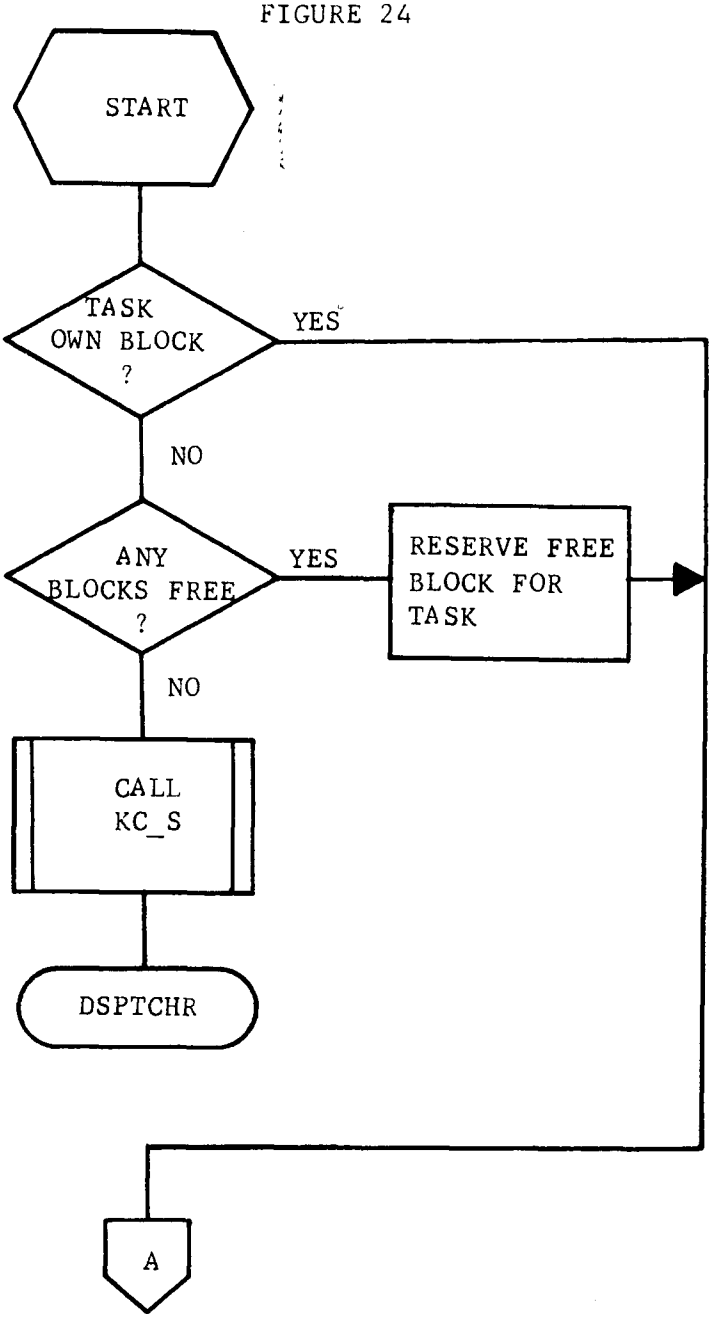


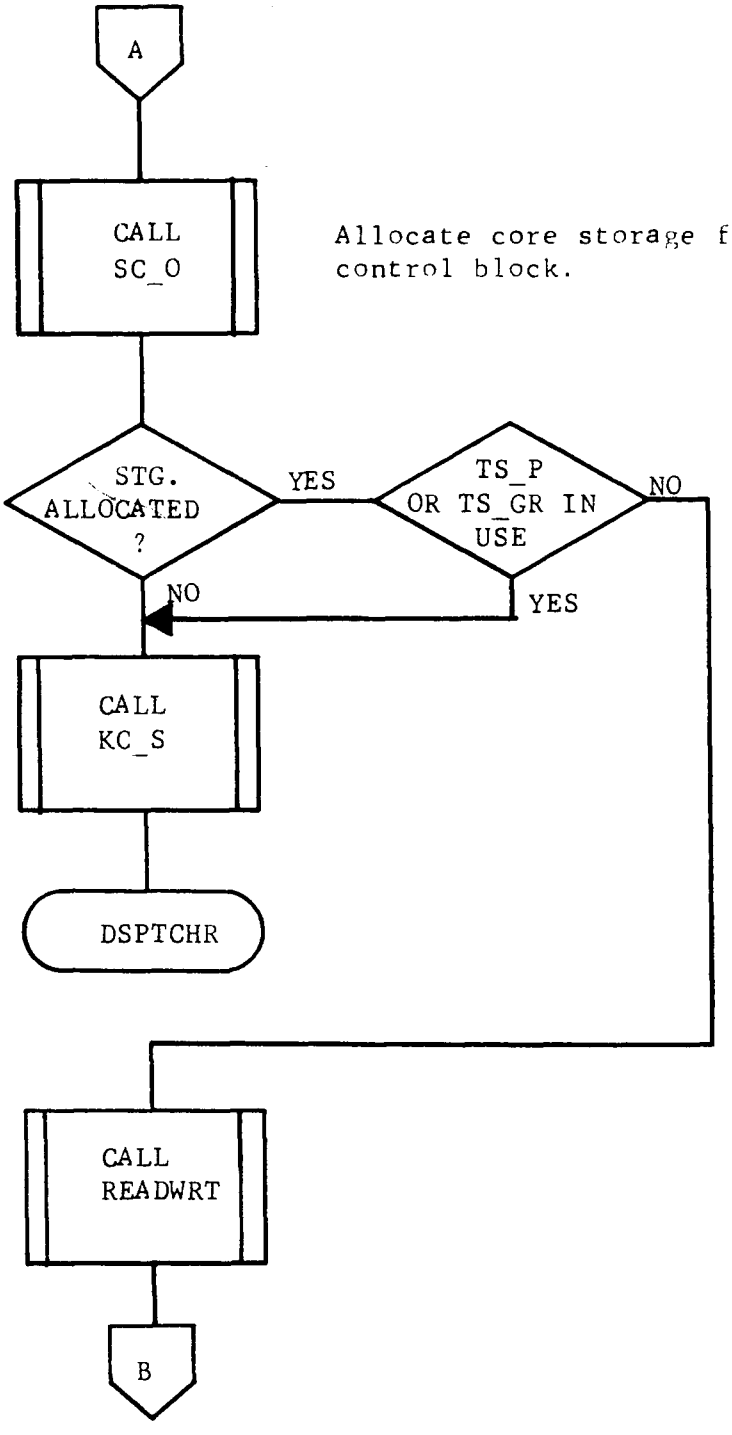

B
E
STG. OBTAINED ?
NO
MORE TASKS IN TABLE ?
YES
C
NO
YES
REMOVE TASK FROM WAIT QUEUE
CALL SC_R
RETURN
A
CUSH. STILL USED
YES
D
NO
RETURN STORAGE CUSHION TO RESERVE STATUS
CALL KC_RS
RETURN

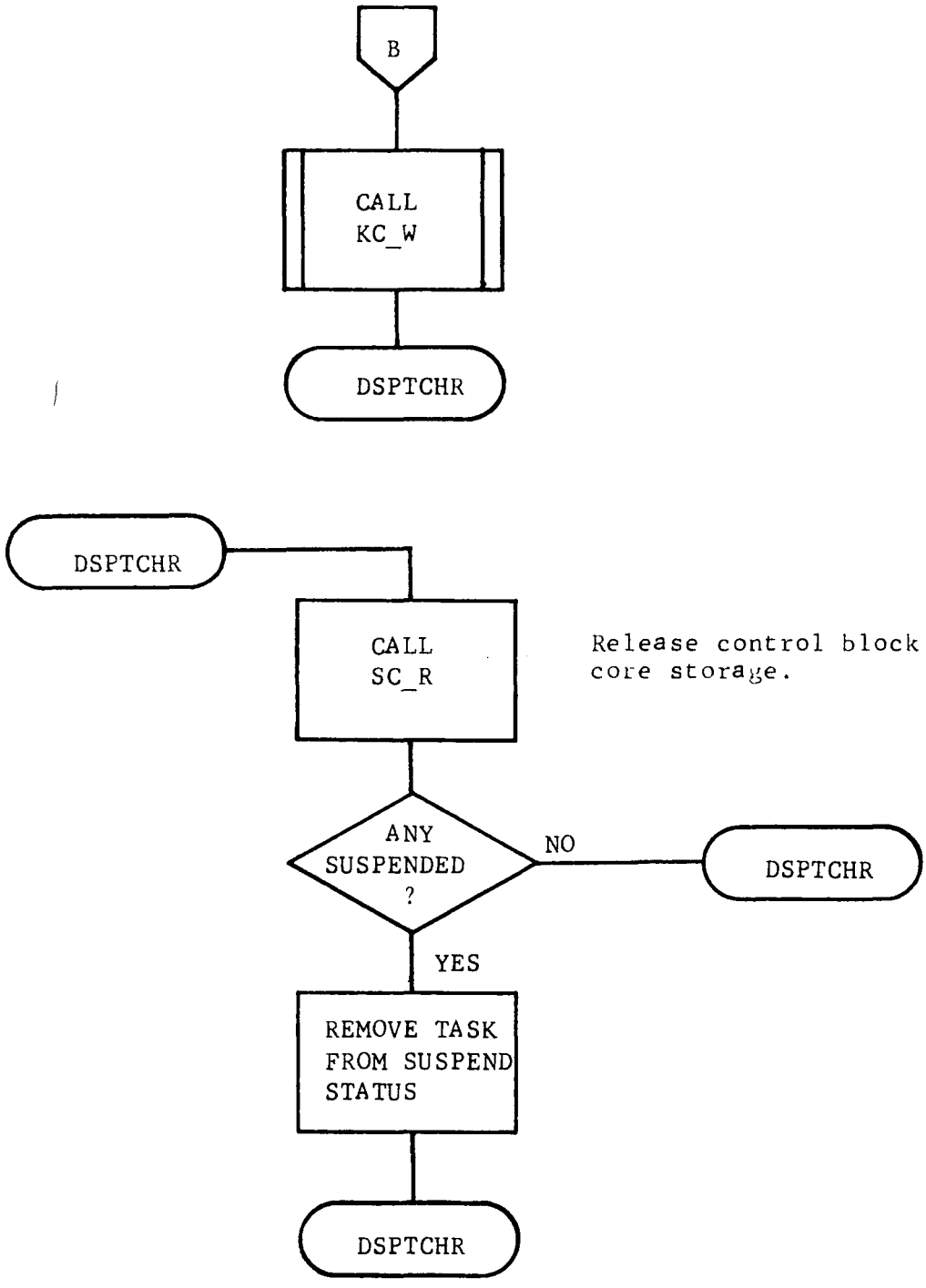

D
FIND PROGRAM IN CORE STORAGE WHICH IS UNUSED AND NOT PERMANENTLY RESIDENT
FOUND ?
NO
RETURN
YES
CALL SC_FS
A

TS_P

FIGURE 24

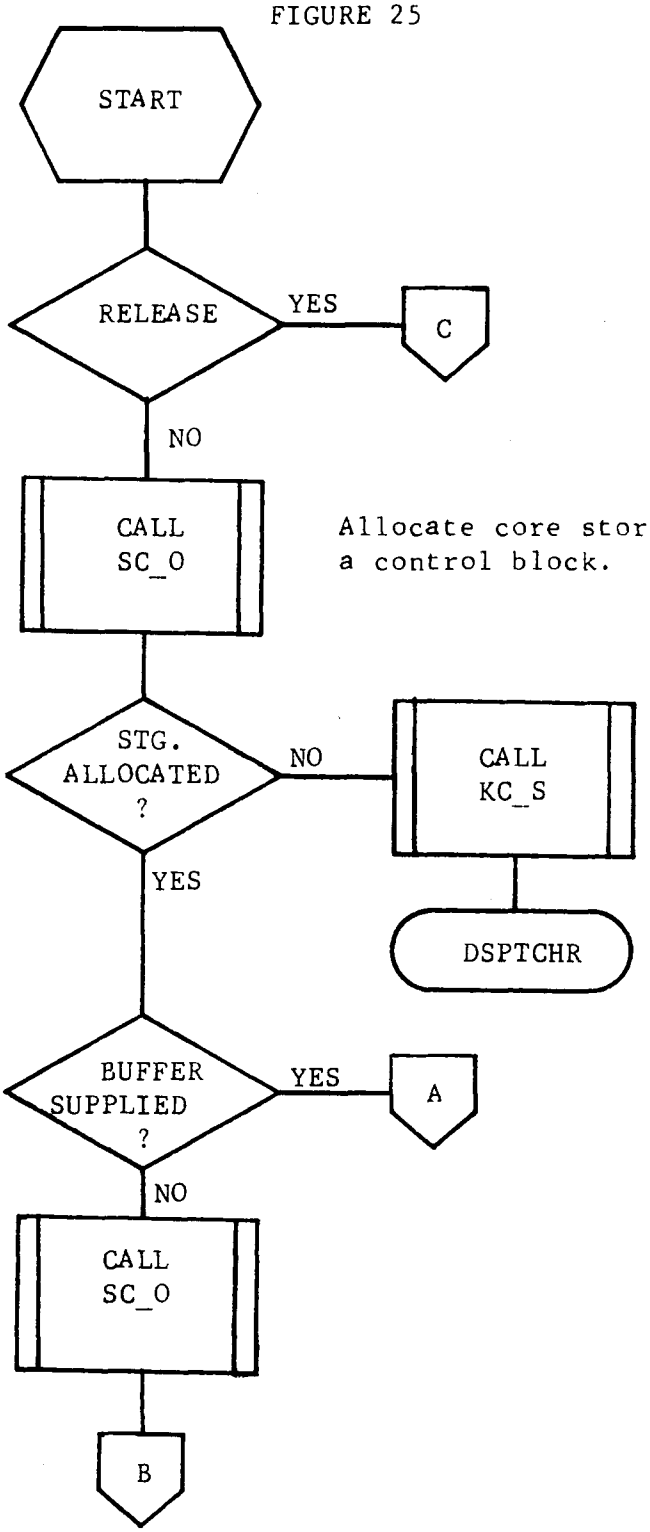


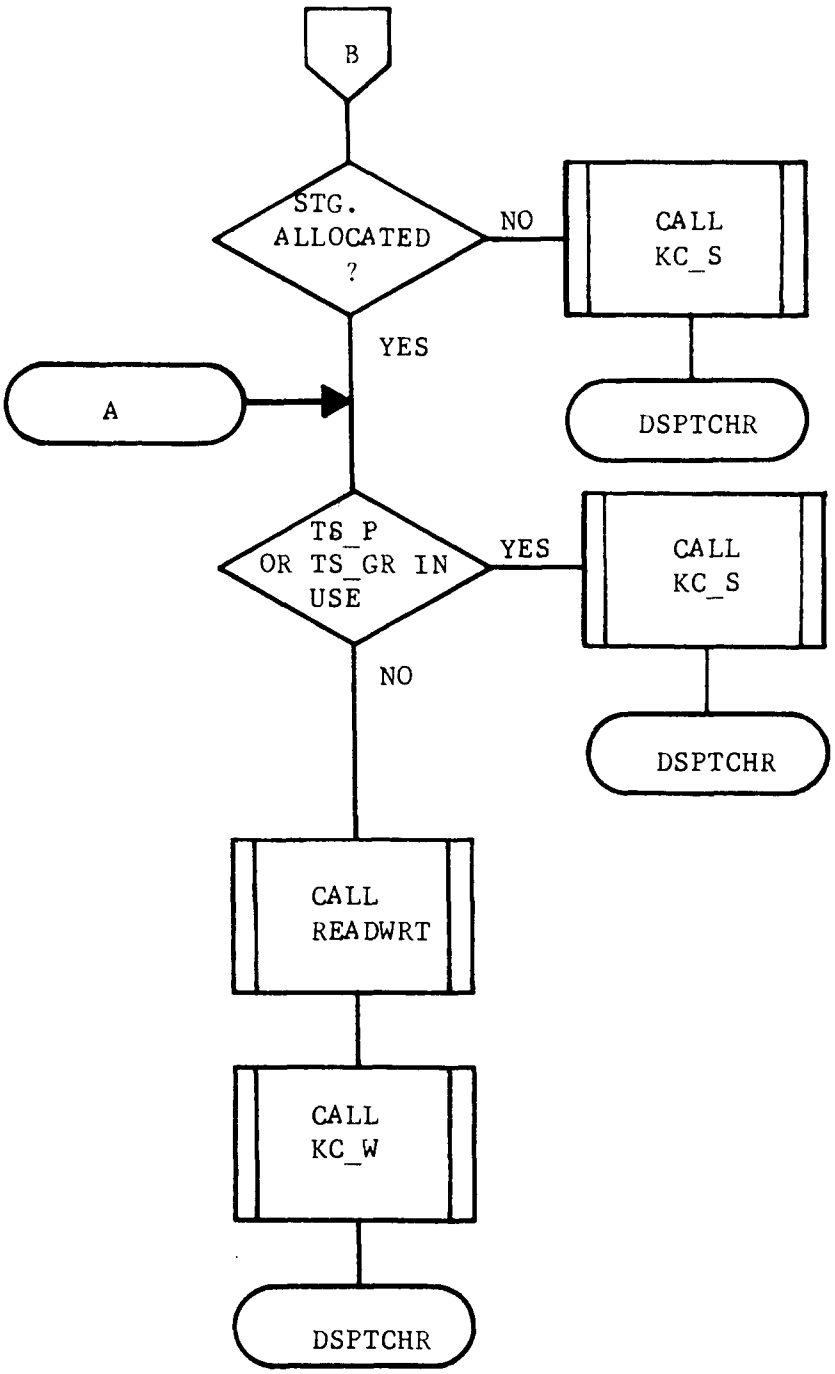

A
CALL
SC_O
Allocate core storage for a control block.
STG.
ALLOCATED
?
YES
TS_P
OR TS_GR IN
USE
NO
NO
YES
CALL
KC_S
DSPTCHR
CALL
READWRT
B

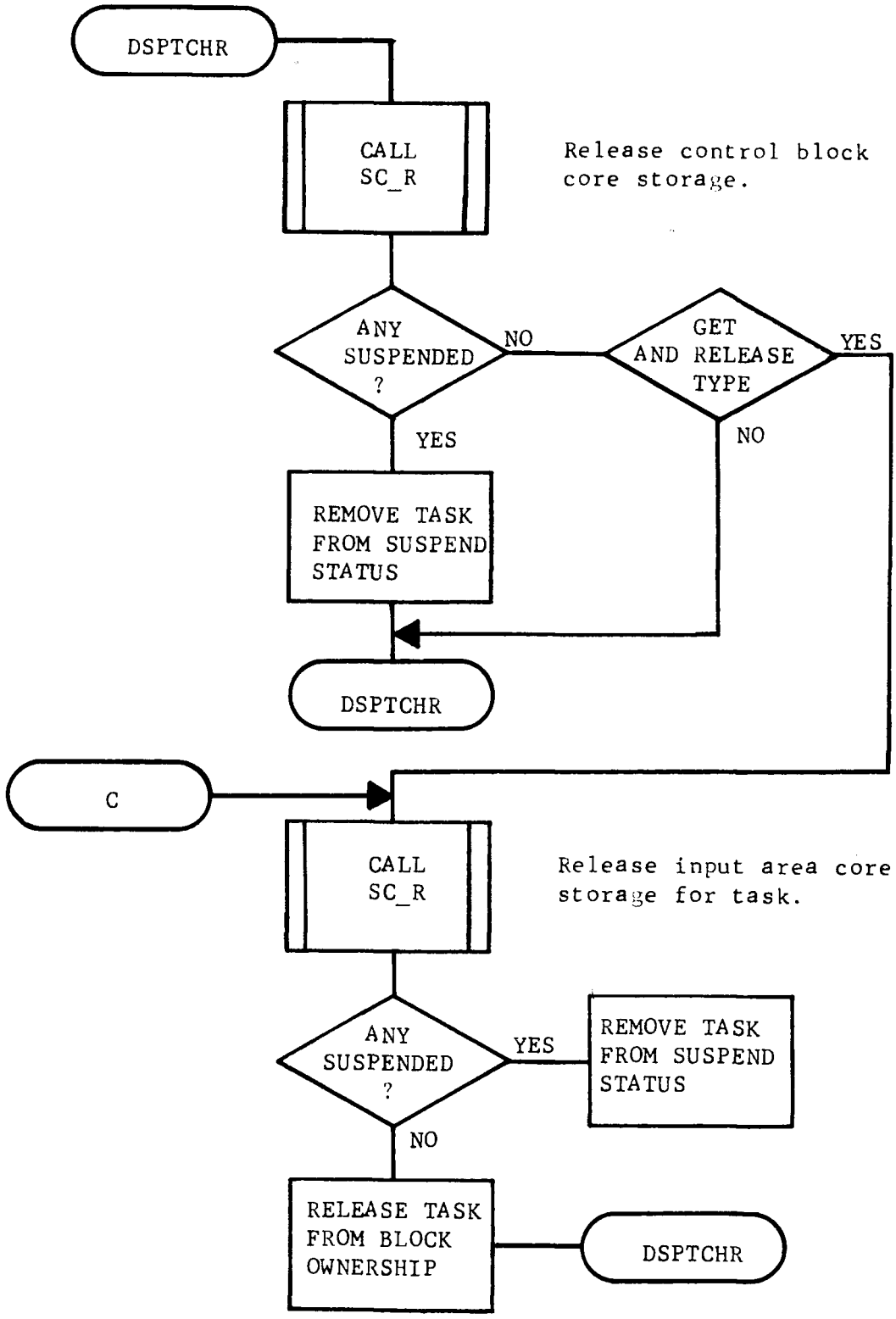

B
CALL
KC_W
DSPTCHR
DSPTCHR
CALL
SC_R
Release control block
core storage.
ANY
SUSPENDED
?
NO
DSPTCHR
YES
REMOVE TASK
FROM SUSPEND
STATUS
DSPTCHR

TS_GR

FIGURE 25

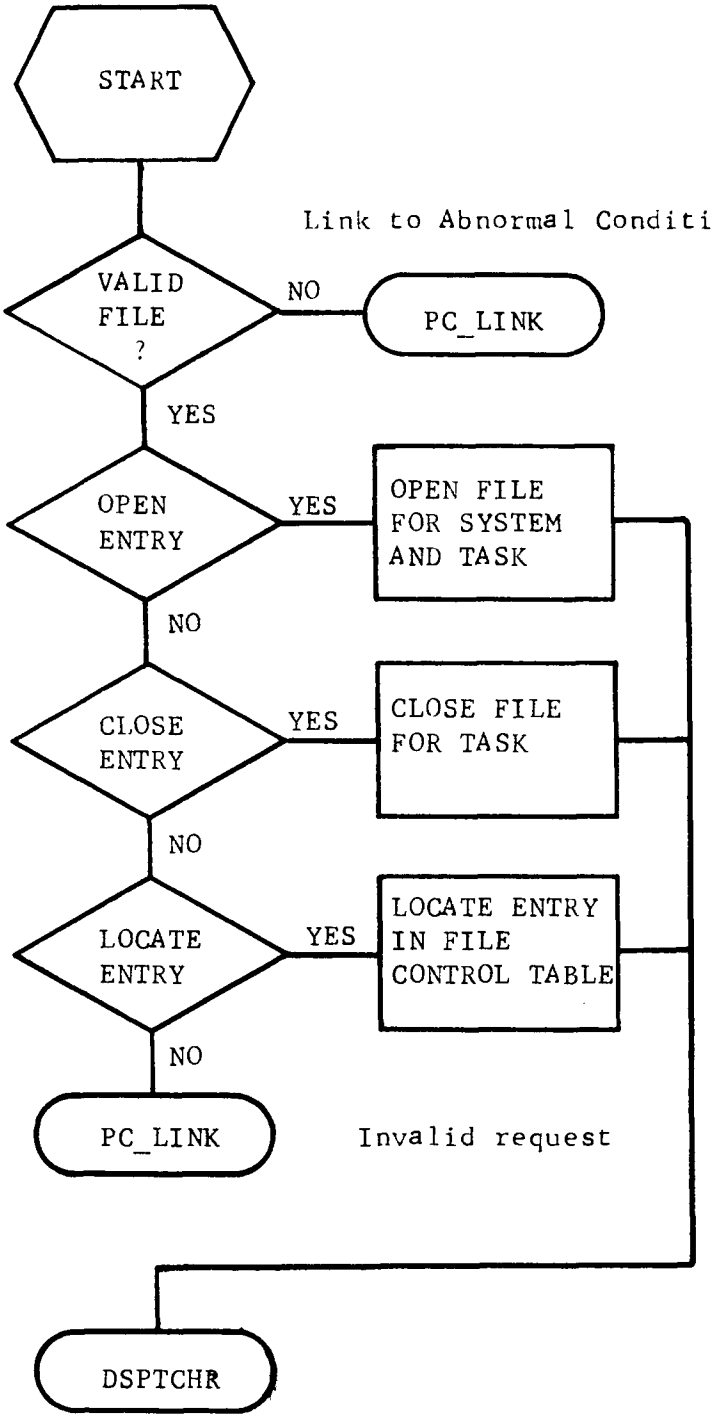



B
STG. ALLOCATED ?
NO
CALL KC_S
YES
A
DSPTCHR
TS_P OR TS_GR IN USE
YES
CALL KC_S
NO
DSPTCHR
CALL READWRT
CALL KC_W
DSPTCHR


DSPTCHR
CALL SC_R
Release control block core storage.
ANY SUSPENDED ?
NO
GET AND RELEASE TYPE
YES
YES
NO
REMOVE TASK FROM SUSPEND STATUS
DSPTCHR
C
CALL SC_R
Release input area core storage for task.
ANY SUSPENDED ?
YES
REMOVE TASK FROM SUSPEND STATUS
NO
RELEASE TASK FROM BLOCK OWNERSHIP
DSPTCHR

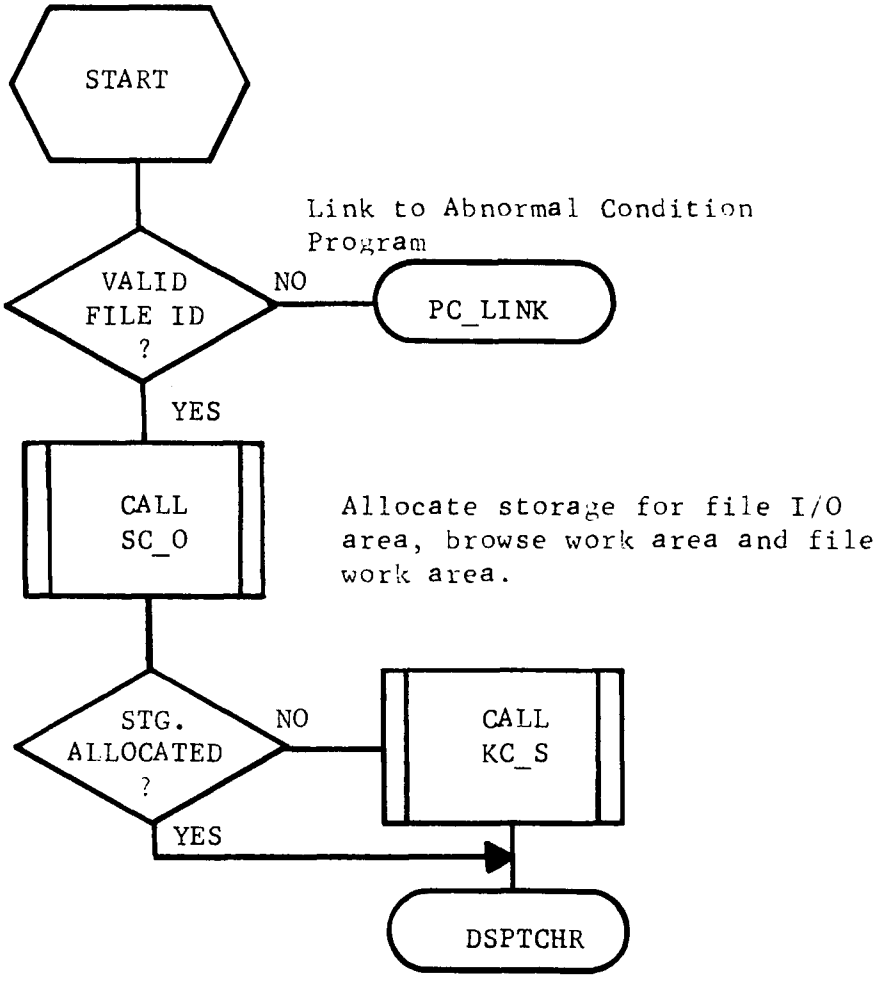
FC_OCL

FIGURE 26



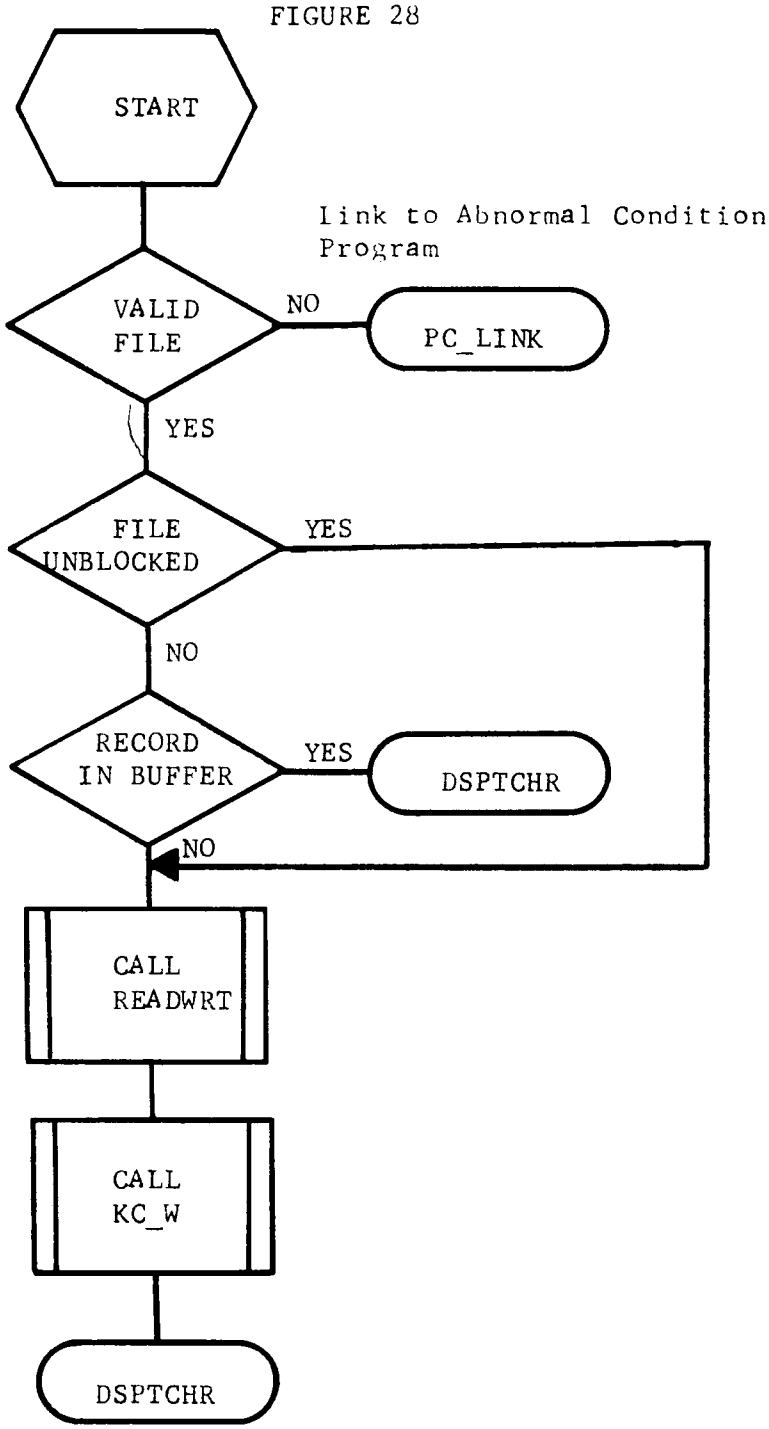
FC_S

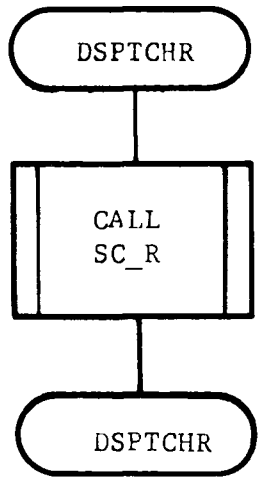
FIGURE 27



FC_GN

FIGURE 28

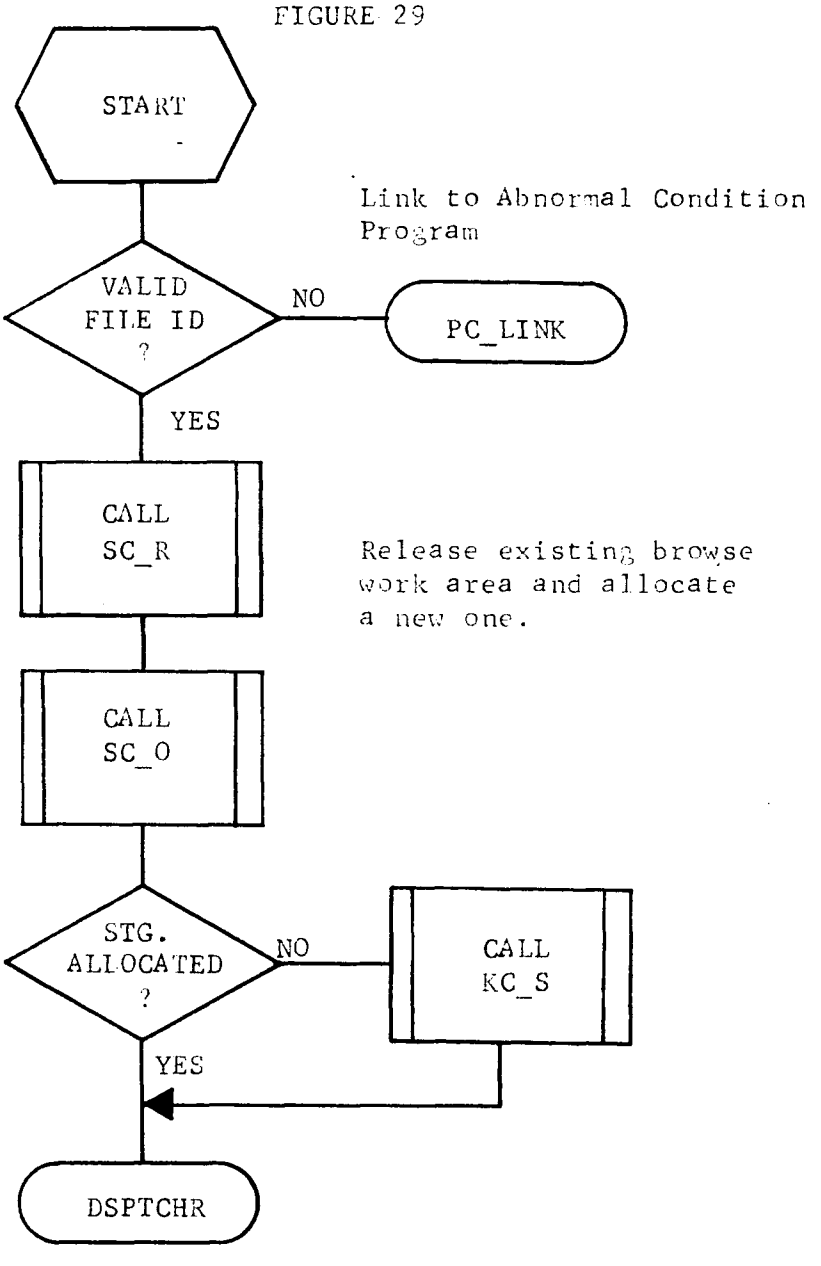





Release file I/O area which was acquired by the task.

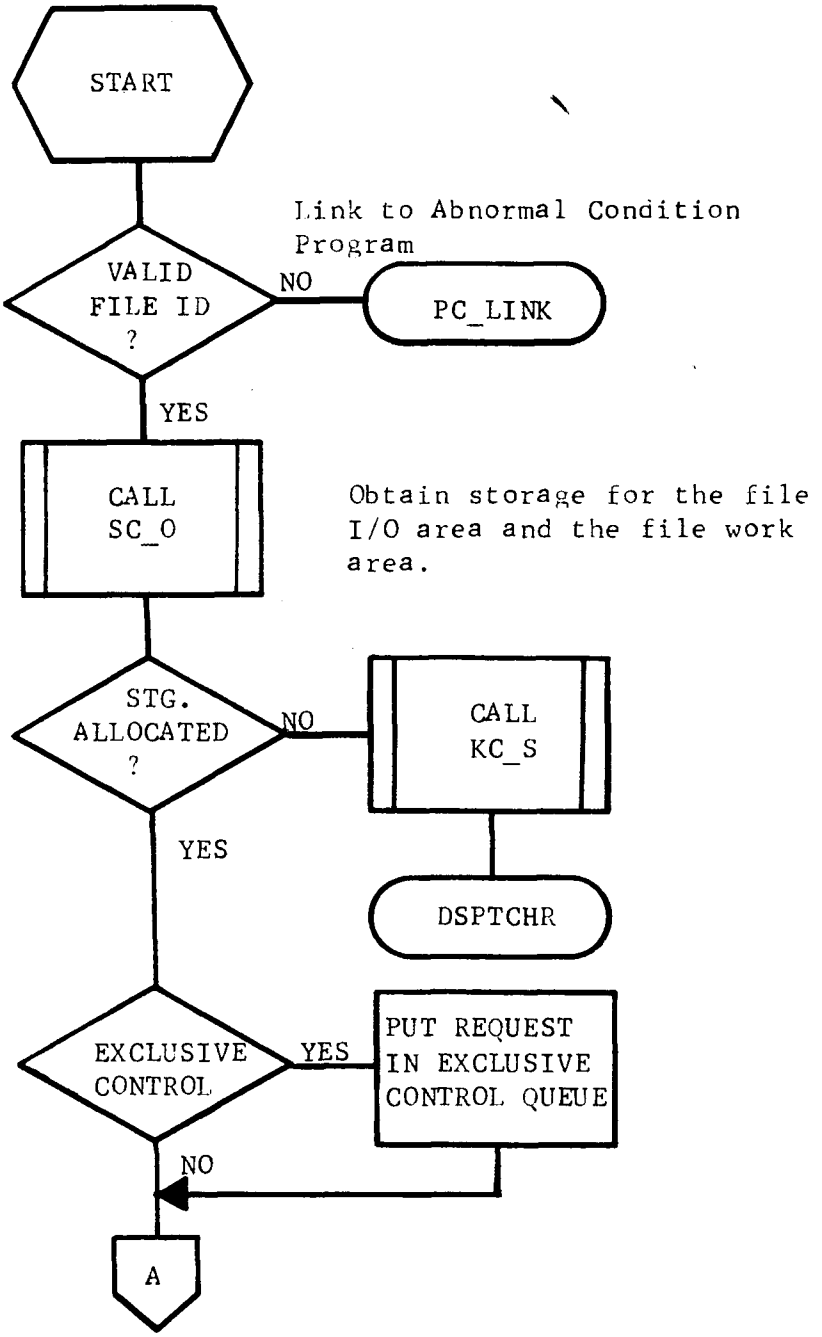
FC_RES

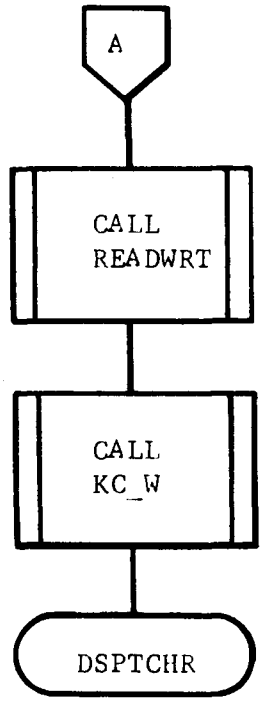
FIGURE 29

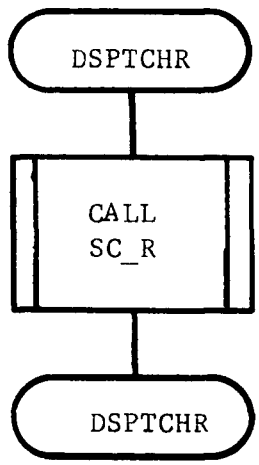


FC_GET

FIGURE 30



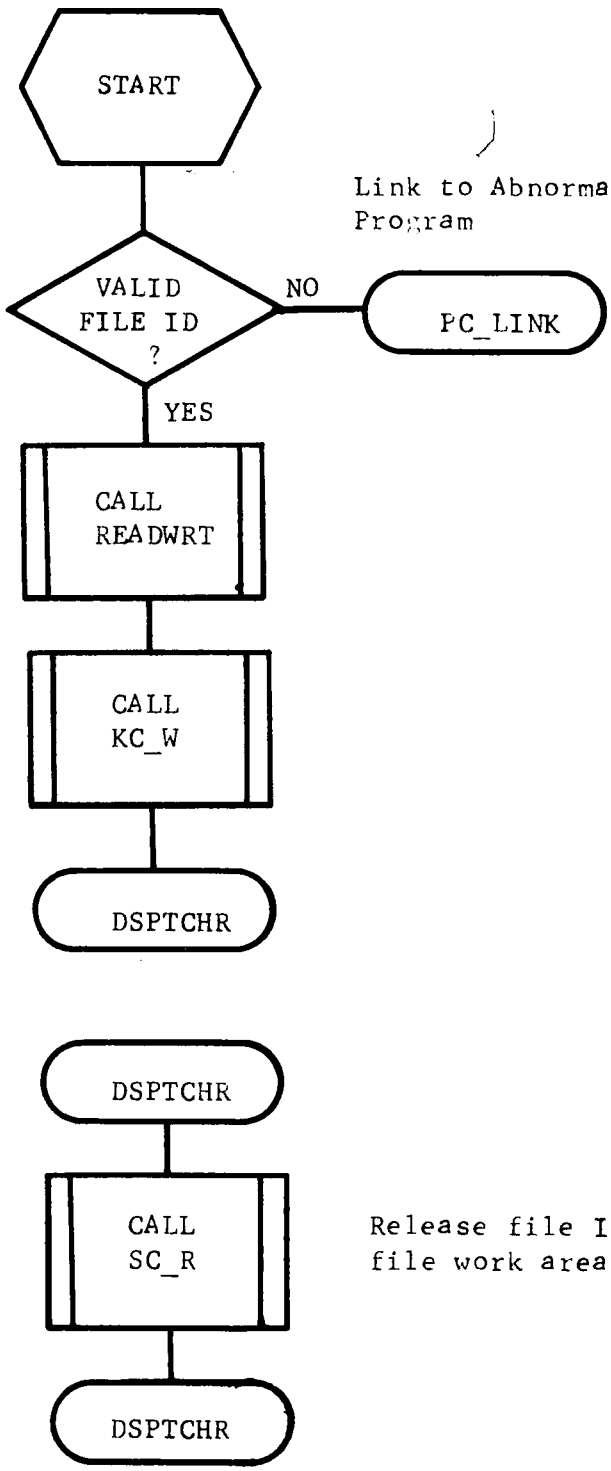



Release file I/O area if not needed.

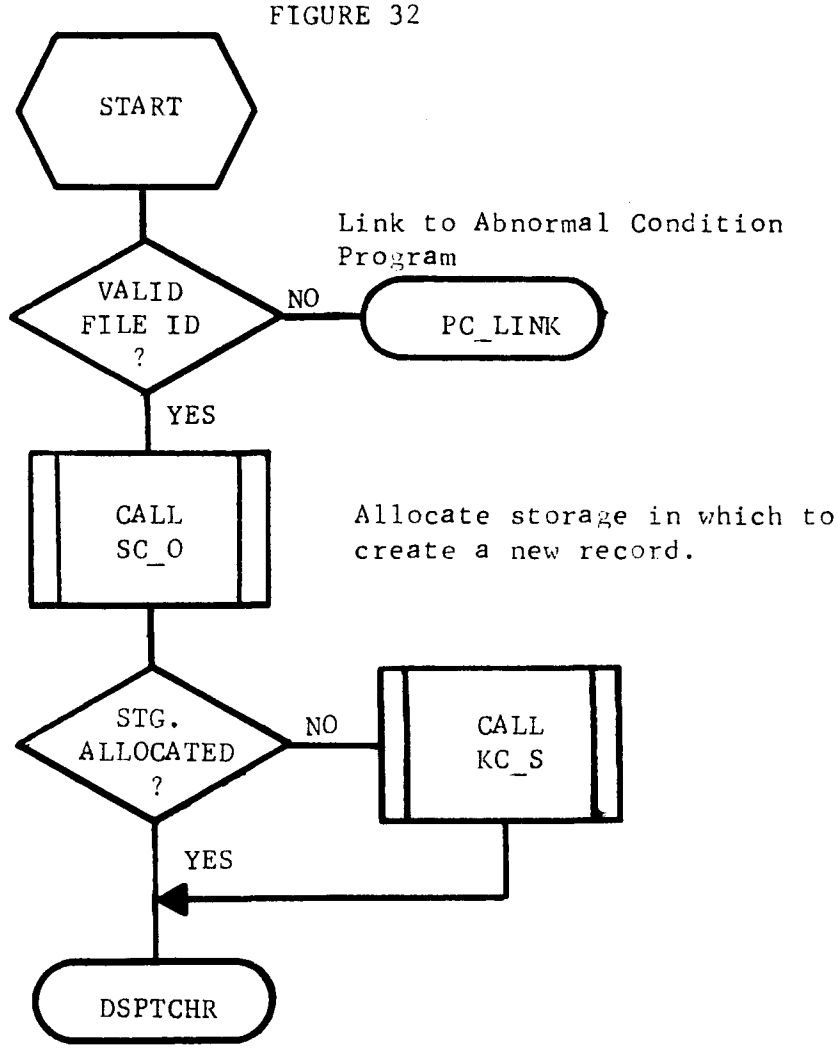
FC_PUT

FIGURE 31



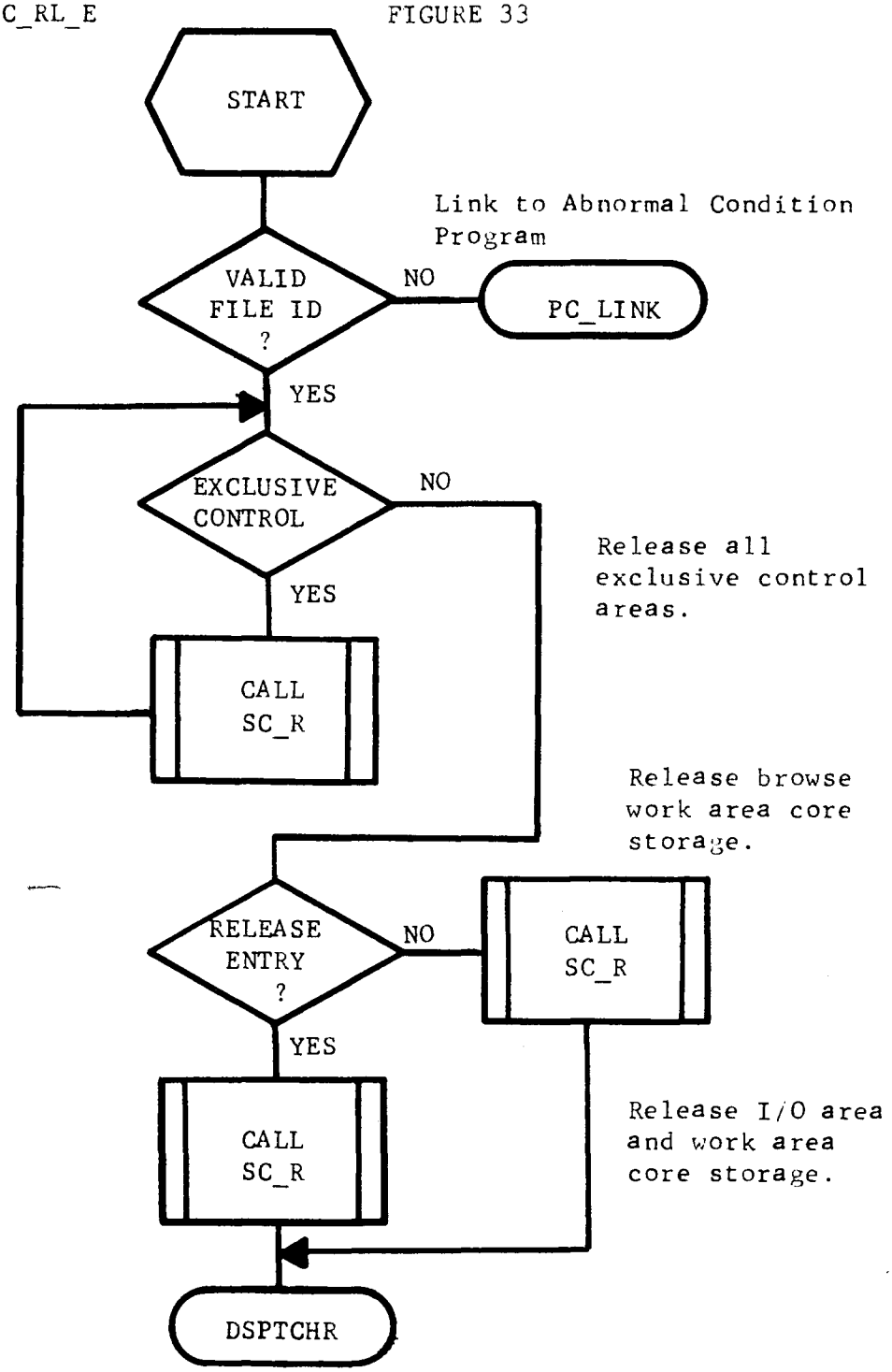
FC_GA

FIGURE 32

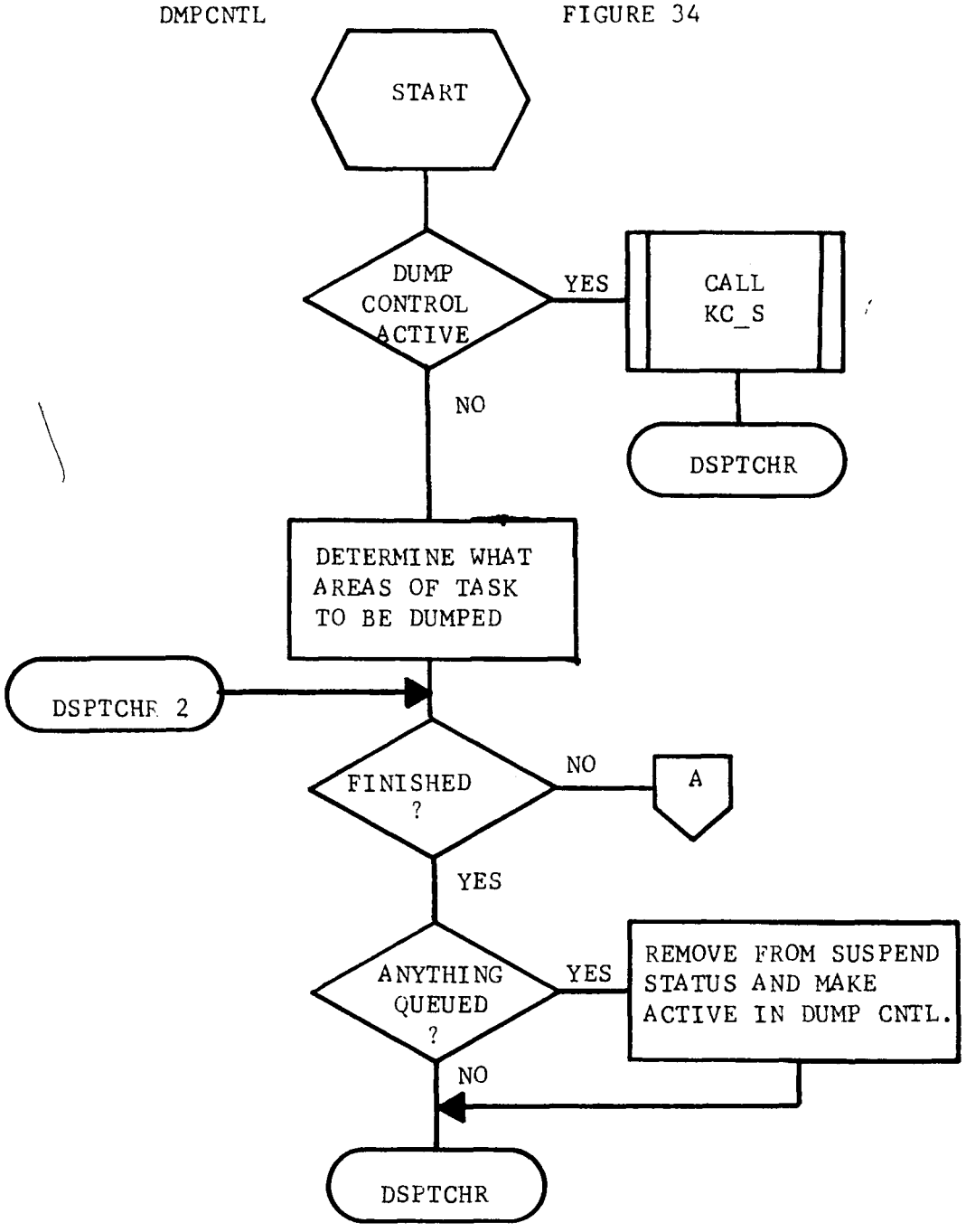


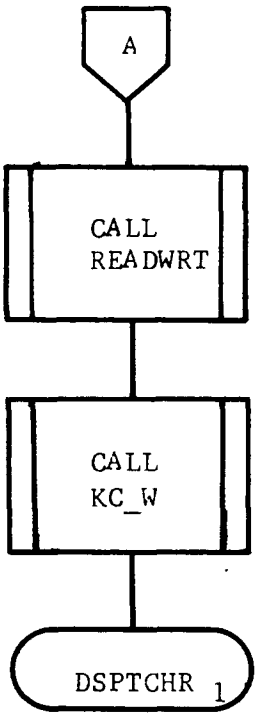
FC_RL_E

FIGURE 33

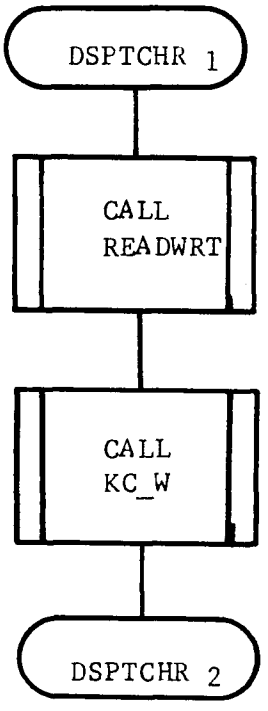


DMPCNTL FIGURE 34





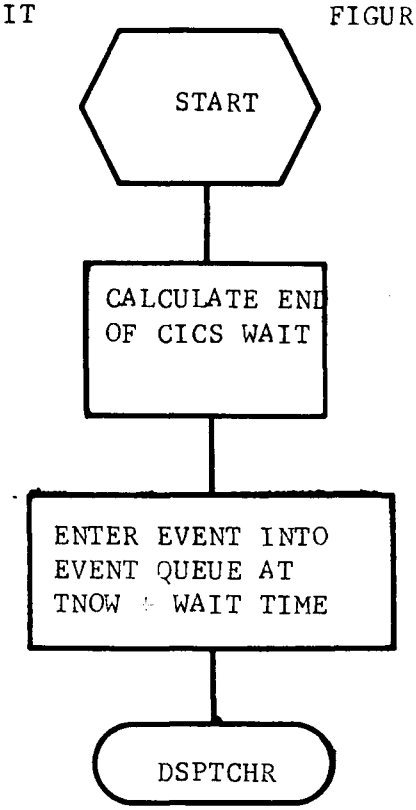

For each area of the task's storage to be dumped, write out an identification record




followed by the storage area.

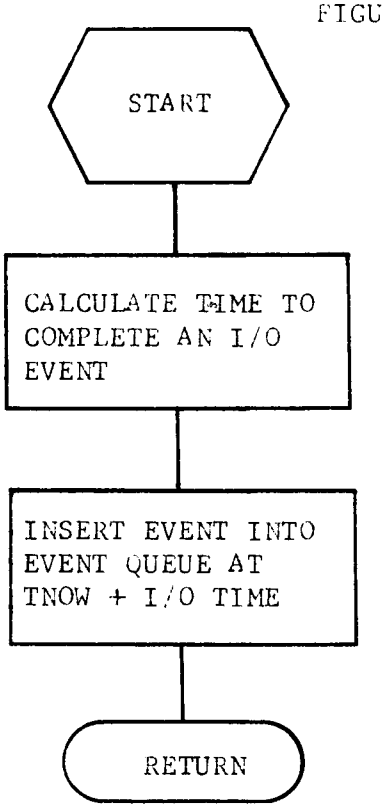
OS_WAIT

FIGURE 35

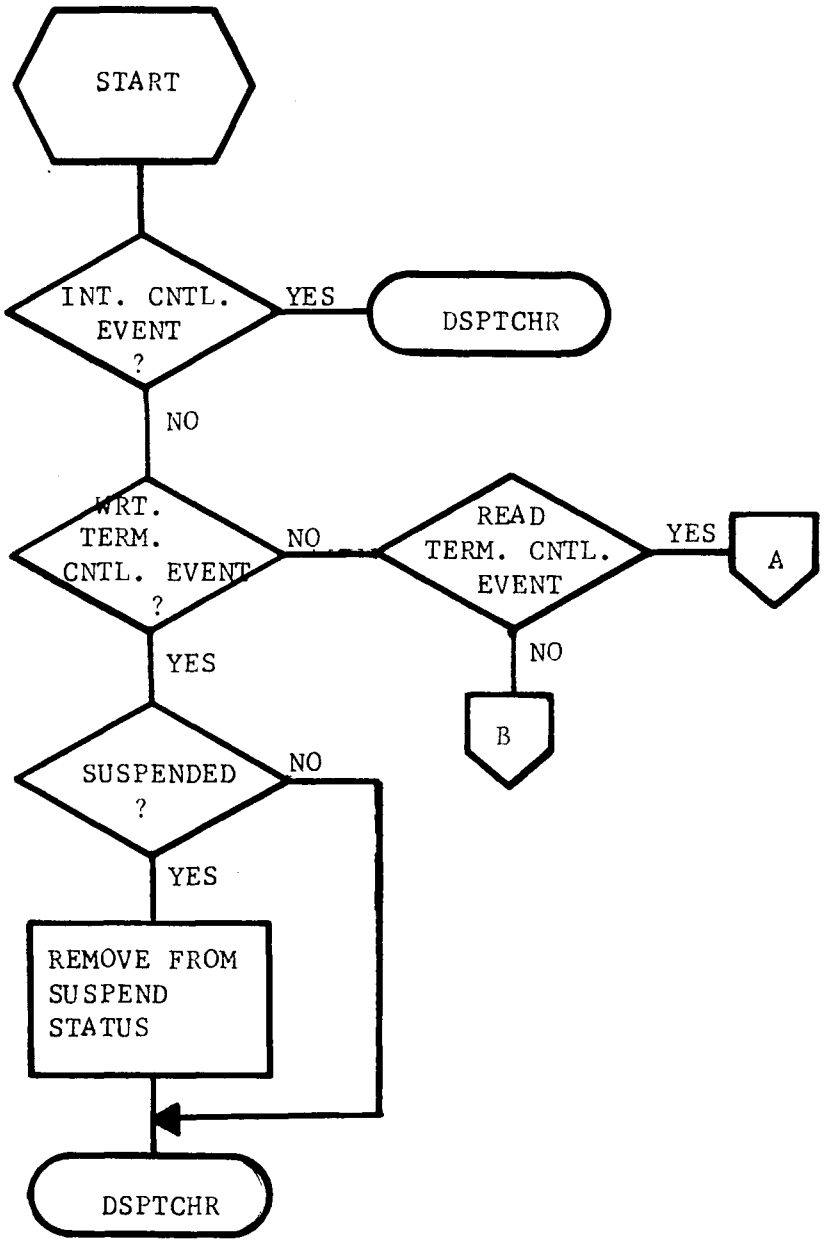


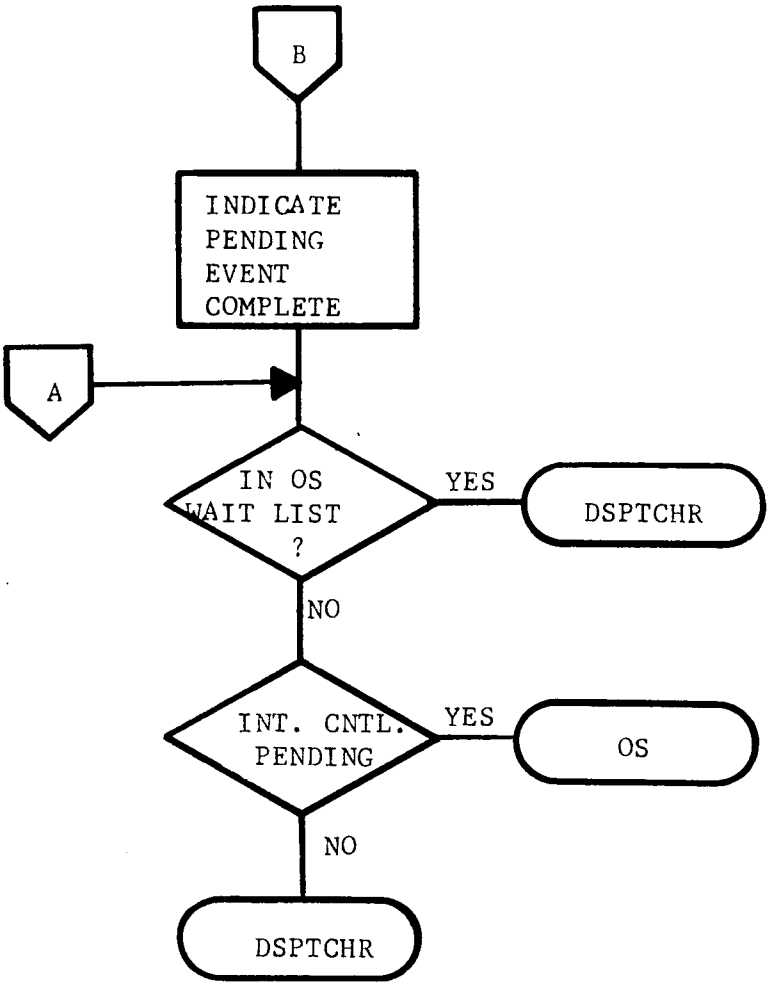

If no tasks are active in CICS, then set an interval control event for the current time plus 2 seconds and exit to the operating system.

READWRT FIGURE 36



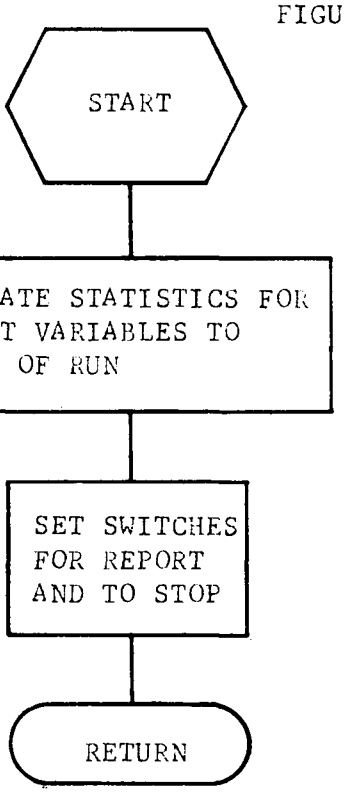
OS_POST FIGURE 37




B
INDICATE PENDING EVENT COMPLETE
A
IN OS WAIT LIST ?
YES
DSPTCHR
NO
INT. CNTL. PENDING
YES
OS
NO
DSPTCHR

END_SIM

FIGURE 38



## APPENDIX F

I am writing this letter to certify that the work done by Donald Hoch for his master's thesis at Lehigh University does accurately model the inner workings of IBM's Customer Information Control System. I make this assertion based first on conversations with Don as he developed his ideas over the past sixteen months and second on my five years full-time experience with the system.

Thomas M Morrison

## BIOGRAPHICAL NOTE

Donald S. Hoch was born in Northampton, Pennsylvania on July 9, 1947, to Evelyn and Sterling Hoch. He graduated from Catasauqua High School and Grove City College where he received a Bachelor of Science degree with a major in mathematics in June of 1969. He also was awarded honors in mathematics. While at Grove City College, he was elected to Kappa Mu Epsilon, a national mathematics honorary society. After graduation from Grove City College, he was employed by Pennsylvania Power and Light Company of Allentown, Pennsylvania as a computer programmer. He is currently employed there as a Computer Systems Analyst. In the spring of 1972 he was married to the former Lucia Marie Cunningham, and they reside in the Allentown area.