# Classroom Model of an Information and Computing System

by

MICHAEL DAVID SCHROEDER

B. A., Washington State University

(1967)

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1969

Signature of Author Signature redacted
Department of Electrical Engineering,
January 20, 1969

Certified by Signature redacted
Thesis Supervisor

Accepted by Signature redacted
Chairman, Departmental Committee on Graduate Students

Thesis
E.E.
1969
M.S.

Classroom Model of an Information and Computing System

by

MICHAEL DAVID SCHROEDER

Submitted to the Department of Electrical Engineering on January 20, 1969 in partial fulfillment of the requirements for the Degree of Master of Science.

## ABSTRACT

This thesis describes the Classroom Information and Computing Service (Clics), a pedagogical computer based information system to be used as a case study in the subject "Information Systems" in the Department of Electrical Engineering at M. I. T. Clics is an abstraction of the Multiplexed Information and Computing Service (Multics) that is currently being implemented by Project MAC at M. I. T. in cooperation with Bell Telephone Laboratories and the General Electric Company. As such, it is an example of a computer utility, currently the most general type of computer based information system that is available. Clics is derived from Multics by a combination of simplifying the mechanisms of Multics and removing some of its more exotic features, and embodies research into ways to simplify the mechanisms of Multics without sacrificing service objectives.

The thesis contains the specification notebook for Clics that will be used in class in the Spring, 1969 presentation of "Information Systems". This notebook specifies both the hardware and control programs of the model system in sufficient detail for students to develop a structural as well as functional understanding of its operation and mechanisms. Projects are suggested which require alteration to the system to increase its efficiency and/or capabilities and allow the students to trace the implications of small changes in a complex system. As the primary case study for the subject, Clics will provide specific examples of the mechanisms and complexities in a general purpose information system.

Thesis supervisor: Jerome H. Saltzer
Title: Assistant Professor of Electrical Engineering

## ACKNOWLEDGEMENTS

Because this thesis is based primarily on a developing system which is not yet fully documented the main source of information has been personal conversations with the people involved in the development effort. I wish to acknowledge the contributions of J. H. Saltzer, R. M. Graham, E. I. Organick, and D. D. Clark in this respect. Work reported herein was supported in part by Project MAC, and M. I. T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01). Reproduction in whole or in part is permitted for any purpose of the United States Government.

TABLE OF CONTENTS

## FOREWORD

This thesis presents the Classroom Information and Computing Service (Clics), a model computer utility which will be used as a case study for a subject in the undergraduate computer science curriculum within the Department of Electrical Engineering at M. I. T. The thesis is divided into two parts. The first part, containing six sections, describes the context for Clics, and discusses such things as its background, development, and use. The second part is the specification notebook for the system that will be used in the classroom to present the model to students.

## PART I

### Introduction

The Classroom Information and Computing Service (Clics) that is described by this thesis is a model of a computer utility that is to be used as a case study in the classroom. As a first step in defining more precisely what Clics is, the undergraduate subject in which it is to be used is described.

At M. I. T. an elective three subject sequence is being developed in computer science within the Department of Electrical Engineering. The subjects are "Programming Linguistics" which considers linguistic constructs for specification of algorithms, "Computational Structures" which discusses hardware and software means of implementing these, and "Information Systems" which considers complexity in information systems. It is for the third subject in this sequence, designated 6.233, that the Clics system is to serve as a case study.

The description of 6.233 in the Institute's general catalog [1] lists the specific topics covered by the subject and is thus a good indicator of its scope. Also mentioned is the role of a model information system, the role to be filled by Clics. The catalog description reads:

> Sources of complexity in information systems, with particular emphasis on problems arising when a single information system serves a community of users. Contrast of problems intrinsic to purposes of use with those related to technical limitations.

> Study of specific design objectives such as reliability, maintainability, information storage, controlled information sharing, ability to withstand change, and ease of administration. Effect of these and other design objectives on system implementation demonstrated in a model system. Comparison and contrast of model system with current state-of-the-art systems, including both general purpose community computing facilities and information systems dedicated to a special task such as telephone line switching or air traffic control.

The term "information system" is intended to include any system which stores and processes information. Thus a library, a telephone system, and a computer system are all valid examples. The purpose of the subject is to describe some of the complexities of interaction that appear when a group of individually understood components are put together to form an information system, and to indicate some of the techniques for dealing with these complexities. The desired result is to prepare students to evaluate information systems with which they may come into contact, and to prepare them to assist in the design of new information systems should the opportunity arise.

A brief outline of the approach used in 6.233 [2] further clarifies the role of Clics. The subject begins by considering information systems in a general way, using the examples of a library, an airline ticket reservation system, and a community computer facility. From each of these examples is developed a list of objectives with the apparent intent of describing ways to derive the systems from the objectives. It is then pointed out that, while systematic approaches to synthesizing a system from its objectives are beginning to appear, a general technique for doing so does not exist. At this point the Clics system is introduced and the specification notebook made available, and the subject split into two streams. One continues the conceptual approach already begun while the other considers the model system, attempting to construct the connections from Clics to the set of objectives that appeared to be desirable in an information system from the previous examples. The second stream actually constitutes a laboratory for the subject. Homework problems based on the model are assigned and each student is given several projects involving modification of the model system.

The case study technique is a useful vehicle in 6.233 because, as in any subject where a formal theory is not available, the ideas here are elusive, difficult to formulate, and difficut for students to grasp. Providing a specific example in which students can observe a set of mechanisms based on the ideas presented incorporates into the subject a piece of solid ground. At the same time, however, the obvious danger that the students will interpret the purpose of the subject to be the presentation of the mechanisms of the model system is present. Because the model provides relatively solid ground, they will tend to grasp it too firmly. The solution to this problem lies entirely in proper emphasis.

That the model system chosen for the case study in a subject on information systems is a computer utility results from the fact that computer utilities are the most general kind of information systems presently available and they contain many clear examples of mechanisms directly related to specific objectives.

There are several precedents for using a case study when teaching subjects related to complex systems. One successful model is the Classroom Assembly Program (CAP) developed by Corbató, Poduska, and Saltzer [3]. It presents an easily understood mechanism for performing the basic operations of typical computer language translator programs, and has been used in several computer programming subjects. Graham's Instructional Translator (Instran) [4] approaches the complexities of compiler programs in the same manner, and has also been used in class work. The SAMOS computer developed by the School Mathematics Study Group at Stanford University [5] allows students to become familiar with the capabilities of a typical computer as a step in teaching them the art of programming. This computer model has been simulated on several machines, and students allowed to construct and run programs on it. Sherman has described a model of a batch type operating system [6] (both hardware and control programs) for the same purpose, to teach programming. Unfortunately, none of these are applicable to 6.233, for each is an example of a very restricted class of information systems. Extending the technique and constructing a model of a computer utility, however, is an obvious approach to the topics of 6.233.

Up to this point we have considered the subject in which Clics is to function as a case study, and have described the role of the model system in the material to be presented. We now turn to the model system itself and consider briefly the scope and basis of Clics. Clics is a model of a computer utility. It is a system providing computational ability and information storage to a community of users in much the same way that the telephone system provides communication ability to a community. It is easily accessible and general in scope. The specific background for Clics is found in the Computer Systems Research Group in Project MAC at M. I. T. which has for several years been engaged in the development of large time-shared computer systems. The first product of this effort, the Compatible Time Sharing System (CTSS) which runs on a modified IBM 7094 [7] has been usefully in operation for several years. Currently a more powerful system, the Multiplexed Information and Computing Service (Multics) which runs on a General Electric 645 computer system [8-16], is being developed and implemented in conjunction with Bell Telephone Laboratories and the General Electric Company. Clics is based on Multics. Multics is chosen as a basis because it is the only system in existence which provides an in-depth implementation of a computer utility in an understandable form.

Clics is a model in the sense that it is an abstraction of the Multics system. It is also an extension of Multics, for it embodies research into finding ways to simplify the mechanisms of Multics without sacrificing the service objectives. Clics is a system in its own right, as well, for it is implementable, and would be useful if it were implemented.

Two basic types of abstraction are used to obtain Clics from Multics. As suggested above, simplification of mechanism is the most important. The second is the removal of some of the more exotic features of Multics. In addition, certain areas of Multics to be included in Clics which have not yet been abstracted in detail are temporarily bridged over with overviews. These bridges, which will be replaced later with detailed mechanisms, provide the model with a completeness that allows it to be used in the classroom while development continues in these areas. Each technique will be discussed in more detail later, and specific examples will be given.

In the five remaining sections of Part I we describe a set of service objectives which characterize a computer utility, describe in more detail the derivation of Clics from Multics, discuss the use of Clics in the classroom, describe the specification notebook that constitutes Part II, and offer some observations and conclusions.

## Service objectives of a computer utility

The objectives that result from consideration of various examples of information systems in the first part of 6.233 define more precisely the scope of a computer utility like Multics and Clics, for these objectives describe an information system in its full generality, and a computer utility is intended to be the most general possible computer system. In particular, they specify the kind of service that a computer utility must provide. To more carefully define the notion of a computer utility, then, and specifically to document the service that Clics is intended to provide (and illustrate to the students), we list these objectives and briefly expand each as it applies to a computer utility.

1. information storage
2. multiple users

3. controlled information sharing
4. simultaneous processing of real time inputs
5. specified real time response
6. scheduled availability
7. user self protection
8. ability to expand
9. ability to adapt to change
10. performance measurement
11. resource management

Following is a paragraph expanding each objective briefly.

1. Information storage is a primary objective of a computer utility. Such a system is of little value to a user if he must always approach it with the entire body of information to be considered in hand. The system must allow each user to build up over a period of many uses a history of information that can be easily accessed. The ability to apply to this information a content related organization must be present. Because the utility serves as the primary storage medium for much of the information, the storage must be reliable.

2. A computer utility is a single system serving a community of users. An obvious parallel is the telephone system. The benefits derived there through the use of a single system for many users apply here as well. First of all, each user has available a range and depth of services (some of which are only occasionally used by him) that he could not support if they were all dedicated to his use. Secondly, it allows easy communication between members of the user community. Such would not be possible, for instance, if each had

instead his own small computer, for the situation would be similar to that which would exist if there were several uninterconnected telephone systems.

3. The communication between users that occurs in the environment of a computer utility takes the form of sharing information. If users are to benefit from each other's work to the fullest extent possible the computer utility must allow each user to share in a controlled manner his body of stored information. The same kind of sharing occurs when users execute service programs provided by the computer utility, such as compilers and library routines. Control is important, for many users need to maintain private information. Company payrolls, budget reports, and grade sheets are examples of private information. These users will make full use of the system only if this information will remain private when stored within it. Required is the ability to define exactly who may use each logical unit of information and in what manner. (The means to allow information to be read, but not altered, for example, is very useful.) The same control may be applied to limit access to the system's control programs.

4. Because many users depend upon a single system, apparent simultaneous processing of real time inputs is required. It must be possible for a reasonable percentage of the total community to use the system simultaneously. Thus, requests of and responses from the computer utility need not be coordinated by the users.

5. If the time required to obtain a specified response from the computer utility varies widely it will discourage system use

and thus reduce the effectiveness of the system. An additional benefit is derived if this response time is small enough to allow interactive use.

6. A computer utility must also be available on a prescheduled basis for much the same reason. The ideal schedule from the point of view of the users is 24 hours each day. Careful integration of system maintenance into the operation of the system can make this goal realizable.

7. The objective of user self protection recognizes that users make errors that can inflict self damage. The computer utility must minimize such damage.

8. The user community of a computer utility will tend to grow larger if the system is functioning properly, and the per user demand on the system will increase. Continued service under these circumstances requires that the ability to expand gracefully be an objective. Faced with the prospect of a system becoming overloaded in the predictable future, users will take their work elsewhere, if possible.

9. For the same reasons the ability to adapt to change must also be included in the list of objectives. This includes both changes in the quality of the user community demand (the need for a new compiler, for example) and advances in technology. It must be possible to incorporate new or improved hardware and control programs into the system with minimum disruption of service.

10. Enough meters must be built into the system so that proper account may be made of system resources. Only by measuring usage carefully can the need to expand be determined and can users be charged for

the resources that they consume. Such measurements also allow the system to be tuned to give highest performance.

11. Objectives 2 and 4 together imply that decisions must be made between users competing for system resources. Optimum system performance demands careful management of these resources.

The objectives taken together define a system with general computing and information storage ability that is well behaved and convenient to use. Making computers available in this form is certainly a step toward their general use and acceptance. It should be emphasized that, while the concept of a computer utility is absolute, the specific set of objectives presented is not unique. There are other cuts of the ideas that produce a different set describing the same phenomena.

## Derivation of Clics from Multics

We now return to discuss in more detail the derivation of Clics from Multics. Fundamentally Clics is an abstraction of the Multics system, and uses the same methods to meet the same general service objectives. Multics, however, is very large and very complex, and many simplifications are made to produce a model that is (hopefully) digestable by students in the time of a semester. The two methods of abstraction that are used (as mentioned in the Introduction) are simplification of mechanisms, and removal of some of the more exotic features. Following, each is considered more carefully and specific examples are provided.

The first is the most interesting, for it embodies basic research into ways to accomplish the goals of Multics with simpler mechanisms than used by Multics. A good example is found in the mechanisms used by each to provide information storage. Multics uses a multilevel memory system incorporating core, drum, and disk storage devices. Fixed sized blocks of users' programs are paged from the drum to core on demand, replacing the removable blocks in core which have gone the longest without being referenced. The replaced blocks move from core to the drum. The same sort of paging occurs between the drum and the disk (via core), though less often. All paging is transparent to the user programs, which always reference their address spaces with a two dimensional address composed of a segment number and a word number. It is not transparent to the system control programs, however, and in fact enormously complicates them. Below a certain level the control programs have the responsibility of keeping track of the location on the storage devices of each block of information, initiating the transfer of blocks between devices, and updating the tables which allow the hardware to translate the two dimensional addresses presented by user programs into absolute core addresses. The contents of the address translation tables change, of course, each time information moves from one place to another. In effect, the control programs have the responsibility of making the multilevel memory system appear as a single level system. The hardware assistance in this matter is confined to performing the address translation from two dimensions to an absolute core address using an in-core segment map associated with each user program and an in-core page map associated with each segment, and to generating faults when a two dimensional address specifies information not in core. This design, of course, was constrained by the available hardware.

In Clics the control programs are greatly simplified by moving as much as possible of the Multics mechanism described above into the hardware. The Clics hardware memory system consists of identical memory modules, each containing words that are individually addressable. These modules provide the entire memory of the system. The memory is seen by the processors and input/output controllers as having a single level, for all words can be accessed with a single sequence of binary addresses. The control programs are relieved of the burden of making a multilevel memory appear as having only one level. It is never necessary for the control programs to initiate the transfer of blocks of information from one device to another or perform the attendant bookkeeping, as in Multics. Memory is still viewed as containing fixed sized blocks of information, but this is for organizational purposes only. Aside from the lowest levels of the storage management subsystem which construct segments from memory blocks by linking them together with page maps, the control programs use two dimensional addresses almost exclusively, just like the user programs. The most important aspect of this scheme may be summarized by saying that each word of information in the entire Clics system has a unique apparent address that does not change. As can be predicted, the result is that the control programs of Clics are much simpler than the corresponding programs in Multics, especially those concerned with storage management.

The specification notebook for Clics does not describe in detail the mechanisms within each memory module, but such memory modules are currently technologically possible. One approach to their construction would be to include in each several layers of progressively slower but higher capacity storage devices, and use automatic hardware paging. (Automatic hardware paging used in the IBM 360/85 system [17].) Thus information currently being referenced would reside in the smallest and fastest memory device while information not being referenced would automatically move to slower and larger devices. Each such memory module could contain many million words of storage, and would preserve the unique correspondence of address and data word when viewed from the outside. A specific address would always retrieve the same information, regardless of on which storage device within a memory module it happened to reside at the time of reference.

Before leaving this example a practical result of the research that went into describing this aspect of the Clics system should be noted. Partly because of the great reduction in complexity achieved in the control programs of Clics with this scheme, the control programs of Multics are currently being re-evaluated to find ways to further isolate the mechanisms controlling paging so that the remaining control programs may view memory as one level.

There are many other examples of simplification of mechanisms in Multics to obtain the corresponding mechanisms in Clics. Below are listed a few more. In each case the Clics mechanism is described first, followed by the corresponding Multics mechanism in parentheses.

. Rings of protection as described in Graham's paper [18] are implemented in Clics by including in each Clics processor a ring register which is compared on each reference to memory with the protection information associated with the referenced segment. (Because the GE-645 processor lacks a ring register, Multics implements rings of protection by maintaining a separate definition of a program's address space for each ring that it occupies.)

. The Clics system uses a 64-bit word length which allows full two dimensional addresses to be contained in a single word. (The 36-bit word length of the GE-645 system forces the use of word pairs in Multics to contain two dimensional addresses and other complex addressing conventions.)

. The interrupt hardware of the Clics system embodies only one level of interrupt and allows processors to accept interrupts only between the execution of instructions. (The Multics interrupt hardware has multiple levels of interrupts and allows processors to accept interrupts at several points during the execution of an instruction.)

. The instruction set of a Clics processor is designed so that any instruction causing a fault can be re-executed from the beginning after the fault is repaired. (The GE-645 processor instruction set requires that faulting instructions be restarted at the point of the fault. This requires that more complete processor state information be saved at the time of the fault than in Clics.)

. In the Clics processors index and pointer registers are functionally combined into a single set of general purpose registers. (Both index and pointer registers appear in a GE-645 processor.)

. The instruction set of each Clics processor includes a special CALL instruction which saves all machine conditions, performs any necessary procedure stack switching, and automatically changes the ring in which the processor is running when necessary. (Multics uses a several instruction calling sequence, and uses faults generated by calls across ring boundaries to interpose into the call a procedure that switches stacks and redefines the address space to correspond to the new ring.)

All of these features of the Clics hardware make possible significant reductions in the complexity of the Clics control programs, as compared to the corresponding Multics control programs.

We now move to an example of the second method of abstraction, the removal of certain of the more exotic features of Multics. Multics has a rather complete user control facility that performs the complex task of balancing the load of the system against the available processor and core resources. This facility also allows for absentee user programs to be scheduled and then be automatically run by the system when the proper time arrives. The Clics system does not include such a facility. It is left out because other portions of the system contain sufficient examples for 6.233, and it is not an integral part of the basic mechanisms of the system.

In other words, the system is complete without it. Load control is a self-contained package that can be added later, if necessary, without restructuring the currently defined Clics system. It may also serve as the source of some good student projects. In general, facilities of Multics are ignored in the model only when they are peripheral to the basic system and when exclusion will not destroy the completeness of the model.

As mentioned in the Introduction, certain areas of Multics to be included in Clics that have not yet been abstracted in detail are temporarily described with general overviews. The overviews provide the model with a completeness that allows it to be used in the classroom while development continues in these areas. Two examples in the version of the notebook presented in this thesis are the areas of system initialization and input/output. Certainly the model system cannot be considered complete unless it includes both. Presently input/output is bridged by an overview of the specialized input/output processors of the hardware system, and another overview which functionally describes an input/output subsystem which only allows a program to read and write a permanently attached typewriter terminal. The details of system initialization are not described yet, either. What is provided is an overview of the system's condition immediately after initialization has occured.

In this section we have seen examples of the methods of abstraction used to derive Clics from Multics. They result in a system meeting substantially the same service objectives as Multics, but which exhibits simpler mechanisms. The specific contents of the Clics system can be determined from the specification notebook presented as Part II of this thesis.

## Use of Clics in 6.233

This section defines more precisely the role of the Clics system in 6.233. It is difficult to specify exactly how the case study will be used, for the subject is still being developed and the specification notebook has not yet been used in the classroom. (Its first use will come in the Spring semester for the 1968-69 school year.) Three uses are seen at this time: students will study Clics as an example of a computer utility meeting the objectives developed for information systems, homework problems will be based on the system, and student projects will be based on it.

The model will be introduced by making reading assignments in the specification notebook and spending one out of the four class hours each week in a laboratory discussing the material read. The emphasis of these discussions will be on how the mechanisms of the model work, why they are necessary, and what alternate mechanisms would work in their places. As the students become conversent with a portion of the model system, homework problems will be assigned that are based on that material. These problems will examine the operation of mechanisms of Clics in given circumstances and will force students to consider the effects of small changes in these mechanisms. Students will also be assigned projects based on the model that require them to make certain modifications or additions to the system with the aim of increasing its efficiency or extending its capabilities. Successful completion of these projects will require a good understanding of the Clics system and careful consideration of the implications of the changes on all system parts. For example, certain table lookups within the system are implemented with simple linear search methods. One project is to replace the lookup mechanisms with a more sophisticated version employing hash coding. A more difficult project is to propose an implementation for an associative memory in the processors to speed the translation of two dimensional addresses to absolute memory addresses. The specification notebook describes some other projects based on the mechanisms described in detail in this version of the notebook. This list, however, is by no means fully developed. More will certainly be added later.

One section of the specification notebook for Clics describes the Clics Implementation Language (CIMPL), a high level PL/I-like language in which the control programs of Clics are to be written. CIMPL was developed for 6.233 by Professors Graham and Saltzer, and D. D. Clark. (The language specification is not included in the specification notebook contained in this thesis, but will be included in that given to students next spring.) Part of learning about Clics will include learning to specify algorithms in CIMPL. Thus both homework sets and student projects can include programming, if appropriate. The study of CIMPL is an integral part of the case study of Clics, because the language contains constructs that are particularly suited to expressing the class of algorithms that appear in control programs for computer systems. Examples of such constructs are data structures, pointer variables, and procedures that are automatically recursive.

As a rough estimate of the weight to be given Clics in 6.233 it is expected that consideration of this case study, both in and out of the clasroom, will occupy one quarter to one third of the time devoted to the subject by each student. This includes time spent doing related homework sets and completing projects.

## The specification notebook

Part II of this thesis contains the specification notebook for the Clics system. The method of description chosen is the same as that used to document Multics. The highly modular specification notebook contains a section for each piece of hardware, each procedure and data base, plus assorted overviews and discussions of system aspects that cut across hardware module, procedure, or data base boundaries. The Clics notebook is written at approximately the same level of detail as that for Multics, but is significantly shorter.

The notebook that is presented with this thesis is a preliminary version. It appears as it will be used in the spring, 1969 presentation of 6.233 (minus the CIMPL language specification) but does not contain all sections that are planned to be included eventually. The dated items appearing in the notebook's table of contents are completed and included in the preliminary version. The remaining items listed there reflect work to be done on the model in the immediate future. The table of contents taken as a whole, however, does not reflect the ultimate specification notebook exactly, for work in some areas has not progressed to the point where the section titles have been specified. In connection with the bridging over described earlier, note that overviews are provided for most unfinished areas so that students have the flavor of the missing sections. In most cases these overviews will be replaced by more detailed overviews when the mechanisms are specified more carefully.

The CIMPL language is used in two ways in the specification notebook: to declare data bases and to define entry points. A brief description of certain characteristics of the CIMPL language will allow these declarations and definitions to be understood in the absence of the CIMPL manual. The data base declarations may be read as PL/I based structures. The only data types are integer, character, bit, and pointer. Integers and pointers occupy a single word. A variable of type bit occupies one binary bit. A variable of type character occupies eight binary bits. Bit and character strings are not allowed, but one dimensional arrays of all data types are allowed. Consecutive bit and character variables in a structure declaration are packed left to right into consecutive bits of a storage word. The declaration for an entry point specifies the entry name as a label, and declares the number and type of arguments in the bound variable list in parentheses. For notational convenience input arguments are named with lower case letters while output arguments are named with upper case.

<u>Conclusion</u>

As a conclusion to Part I the significance of the work reported in this thesis is summarized and the future development of the Clics system is outline. The basic significance of Clics is that it allows the presentation to students of material which up to now has been inaccessible to them. It has been the case that for a student to acquire the ability to evaluate and innovate information systems, particularly computer based information systems, he had to become involved in the design, construction, or maintenance of such systems and slowly build a background of experience from scratch. All too frequently this method resulted in exposure to a restricted class of information systems. Such background was not available in a generalized, condensed form. With primary support from the Clics system, 6.233 will provide an alternate background so that students need not start from scratch in gaining experience. In other words, the model system will allow something to be taught that was not being taught up to now.

A secondary significance of Clics is that it demonstrates the large simplifications of mechanism that are possible in a system like Multics when hardware and control programs can be specifically designed to meet the objectives of the system, and when simplicity of mechanism is viewed as an important objective in its own right.

The claim is not made that Clics is the only model useful in 6.233, or that Clics embodies the only way to meet the service objectives stated as desireable in a computer utility. Although Clics is the primary case study, other examples of information systems are considered in 6.233 (the Electronic Switching System developed by Bell Telephone Labroatories [19], for example). As stated earlier, a model of a computer utility is chosen because computer utilities are the most general type of information system currently available, and the means used in Clics to meet the service objectives are abstracted from Multics because Multics is the only currently available system providing an in-depth implementation of them.

It should also be emphasized that this thesis presents a snapshot of work in progress. The argument that the snapshot is complete enough to be of interest follows from the use of this preliminary version of the notebook in the classroom next semester and the existence of overviews bridging unfinished areas.

The immediate future of Clics includes, of course, its use in 6.233 this spring and the completion of the specification notebook sections specifying in detail the areas bridged over with overviews. As a result of its use in the classroom a comprehensive list of student projects will be defined and debugged. It is also expected that the model itself will be debugged by its exposure to students, and that ways will be found to rewrite specific pieces of the specification notebook to make them clearer.

A concurrent activity will be the production of listings in the CIMPL language for all the control programs of Clics. This has already begun - two undergraduate students are writing the procedures of the storage management subsystem. These listings will be included in the specification notebook as part of the system description. They will be valuable both in allowing students to develop a very detailed understanding of the mechanisms in Clics and in demonstrating the use of a high level language as a means of coping with the complexity of a computer utility.

A compiler is being constructed for the CIMPL language. At the present time it is planned that it will only check the syntax of programs, but thought is being given to constructing some sort of simulator in which the control programs of Clics, written in CIMPL, can be run. This will allow students to actually make changes to the system, run the altered system, and observe the result of their work.

## PART II

### The 6.233 Clics System Specification Notebook

The following pages present the preliminary version of the specification notebook for the Clics system. It begins with a table of contents listing the sections of the notebook.

6.233 Clics System Specification Notebook

Table of contents
Date: JAN 13 1969

Date: DEC 16 1968

## Identification

Student projects

## Purpose

This notebook section presents possible student projects based upon the Clics system, as it is described in the remaining notebook sections. The projects range from easy to difficult, and by no means exhaust the possibilities. It is expected that students will also be able to suggest projects which interest them. In each case the project is a change or addition to the system that involves altering the hardware or software. Projects are to be written-up as new or changed notebook sections, with flowcharts and/or listings when applicable.

A preliminary report in which the student presents his understanding of the issues involved and problems to be solved, and outlines his solutions will aid in determining if he has a good grip on the problem, and allow exchange of ideas between instructor and student.

The project list in this version of the Clics notebook is not fully developed. A few projects are completely described to provide the flavor of suitable project statements. Others are only identified in a list, or have not been thought of at all.

## Project 1: Access mode indicator execute permit bit

This project is to add an execute permit bit (E) to the format of the access mode indicator that appears in each segment descriptor word, and to modify the processor logic to check this bit each time an attempt is made to execute a word of a segment as a machine instruction. $E = 1$ & $R2 \geq$ procedure ring register of a processor must be true before the processor will retrieve an instruction to be executed from a segment. If it is not true then an illegal procedure fault should be generated.

The suggested implementation is to add an EXECUTE operation indicator to those recognized by the address mapping logic of a processor, and to have the instruction fetching logic retrieve the instruction to be performed with it. The address mapping logic will return the instruction word only if the supplied procedure ring register value properly compares to the R2 ring number in the access mode indicator of the containing segment, and if the contained E bit is "on".

Also included in the project is making the alterations to the procedures and data bases of the storage management subsystem that will allow the E bit to be used. These changes, though minor in each case, are surprisingly numerous.

## Project 2: Address mapping logic associative memory

This project is to add an associative memory to the address mapping logic of the processors that will make it unnecessary to retrieve segment descriptor words (SDWs) repeatedly with repeated reference to the same segment by the processor. This project is not easy, and there are several difficult problems lurking just below the surface.

The general implementation scheme suggested is to define a set of live registers (how many?) that can be quickly searched by the basic address formation logic of the address mapping logic for the currently needed SDW. The intent is that these registers contain the SDWs of the segments most recently referenced by the processor. The search is ordered, with the most recently used SDW being inspected first. Only if this search is unsuccessful is the SDW retrieved from memory. A SDW retrieved from memory replaces the SDW in the associative memory that has gone longest without use, and becomes the most recently used.

Some mechanism must be described for clearing the associative memory. One obvious occasion when this is necessary is when a processor is switched to the address space of another process by changing the contents of the descriptor base register. The same segment numbers may correspond to different SDWs in the new process. This case can be handled reasonable easily, for the simulus for clearing the associative memory comes from within the processor, whose associative memory is to be cleared.

Other cases may be less obvious, and more difficult to handle. For example, each time the fault bit in an SDW of some process (A) is set "on" by the process address space manager of the storage management subsystem executing in another process (B), and process A is concurrently running on a processor, that processors' associative memory must be cleared. This means that the processor on which process B is executing when the fault bit is set "on" must signal the processor on which process A is executing to clear its associative memory.

One solution to these problems may be a machine instruction which clears a processors' associative memory. This could be used immediately after each load descriptor base register instruction, and could also appear as the first instruction of the fault/interrupt interceptor. Thus, the sending of an interrupt to a processor would cause it to clear its associative memory. The ability of a processor to ignore interrupts (run with the interrupt inhibit switch set "on") may cause some problems, however.

Another solution may be to have each processor clear its associative memory automatically as part of the load descriptor base register instruction, and to clear it at the beginning of an instruction cycle if any interrupt present line to the processor has a signal on it, whether interrupts are inhibited or not.

Project 3: Improved memory map and memory manager

This project is to design a new memory map and memory manager (both part of the storage management subsystem) that will not require all empty blocks of memory to be linked together at system initialization time, and that will allow more than one process at a time to assign or release memory blocks. The interface presented by the new memory manager should match exactly that currently described. A suggested implementation is to divide the memory map into several pieces, perhaps one per memory module, and provide a separate lock for each. Thus, as many processes as pieces could simultaneously manipulate the map without interfering with one another

Initially the free blocks of each module would be unlinked, and grouped contiguously at a high address end of the module. Assignment of blocks would be sequential. Released blocks would be linked by absolute address into a chain. Further requests for block assignment are met by assigning the block at the head of the chain. Only when the chain for a module becomes empty are further blocks from the never-used portion of the module assigned.

## Project 4: Hash-coded system segment catalog search

This project is to replace the current linear search of the system segment catalog (of the storage management subsystem) with a faster hash-coded search. The search is performed each time the segment catalog manager is presented with a segment identifier which may or may not locate an entry in the catalog table. Because of the extreme length of this table, a failure of the linear search may usurp 10 or more seconds of processor time. A hash-coded search would allow success or failure to be determined in an average of two table references (assuming a half full table), greatly increasing the segment catalog manager's efficiency.

Project 5: Variable length descriptor segment and identifier table

Include in the process address space manager (of the storage management subsystem) a mechanism which allows the descriptor segment and identifier table of a process to vary in length as segments are added to or removed from the address space of the process. Currently, both are of fixed length, a practice which either limits the number of segments that may be in an address space or wastes memory, depending on whether they are too short or too long. A more ideal mechanism would automatically adjust the length to circumstances. Existing facilities of the SMS should be used to change the physical length of the segments.

Project 6: <u>Variable length directory segments</u>

This project is to modify the directory format (if necessary) and directory manipulator of the storage management subsystem so that the length of a directory will automatically be increased as branches and directory control list entries are added to it. Currently directories are of fixed length, and an error code is returned if a manipulation overflows the directory being manipulated. An extension of this project is to make directories shrink when branches and directory control list entries are removed. This is more difficult, for the remaining information seldom cooperates by grouping together at the beginning of the segment. Be sure to consider the possible consequence of chainging a directory segment's length in such a way as to alter its absolute base address.

Project 7: Return pointer entry point to the storage management subsystem

This project is to add a path through the SMS (which is callable by a user procedure from outside of ring zero) that will cause a segment to be removed from the address space of a process. The entry point in the user interface manager would be defined by the following CIMPL statement:

```
   remove: entry (segptr pointer, validation_level integer, RESULT bit);
```

The first argument is a pointer to some word of the segment to be removed, and the second is the validation level of the calling procedure. The user interface manager would, after appropriate argument manipulation and verification pass the call on to other modules lower in the SMS that would make certain the call to remove the segment comes from a ring at least as low as the ring from which the segment was included in the address space (record of this must be kept somewhere), and then remove the SDW and inform the segment catalog manager of the removal of the segment. If removal is successful then RESULT is returned "on". If unsuccessful, for any reason, then RESULT is returned "off".

Project 8: Courteous segment deletion

This project is to alter the storage management subsystem so that a segment may be deleted in two ways: courteously or otherwise. The method currently used is the latter case - the deleted segment disappears immediately and is removed from the address spaces of all processes using it. Courteous deletion would allow processes currently using a segment to continue to do so, but would deny access to new processes. The last process removing the segment from its address space would cause it to be physically destroyed.

One method of implementation would include a courtesy switch in the argument list for the delete entry to the SMS. If "off" deletion would occur as it does now. If "on" then the segment's branch would immediately disappear from the hierarchy, but the system segment catalog entry would remain, suitably flagged, disappearing only when its use list became empty. Special problems may appear when courteous deletion is applied to directory segments?

Project 9: <u>Immediately effective access control list changes</u>

This project is to provide the option of making changes to the access control list of a segment immediately effective. Currently, alterations to an access control list have no effect on processes which already have a segment in their address spacer. For example, a change removing the access control list entry giving user A permission to cause his processes to read a segment would currently not affect user A's process which had the segment already in its address space at the time of the change. It would become effective only after user A's process gives the segment back. His processes would not be allowed further access, from that point on. The access control list alteration could be made immediately effective by setting the SDW fault bit in an affected process "on" and forcing the process to reinclude the segment in its address space, under the constraints of the new access control list. Which method of making the change to be used would be selected by inspecting a courtesy switch argument (caller provided). If "off" then the immediately effective method would be used. If "on" then the current (courteous) method would be used.

Additional projects (not written up)

10. add symbolic links to the storage management subsystem hierarchy

*11. define a set of storage management subsystem driving commands (to operate in user ring)

*12. implement storage management subsystem commands

* Projects 11 and 12 need to be more closely defined. Specific commands should be suggested.

## Identification

Overview of hardware organization

## Purpose

This section describes the overall organization of the hardware in the Clics system. The hardware and the control programs are highly functionally dependent, and the hardware has been designed to operate in the specific environment provided by the "software". Thus, hardware features must be viewed in conjunction with the associated control program environment.

## Introduction

The hardware system is organized as an interconnected group of modules. The most important kinds of modules are processors, I/O controllers, and memories. At any specific time, the number of each kind and the pattern of interconnection is fixed. Both, however, may vary over the life of the system. The use of many copies of a few basic modules, and a standard system-wide interconnection allows this variability of configuration in response to changing circumstances.

## Organization

The three major kinds of modules are divided into two groups: active and passive. The active modules (the processors and the I/O controllers) are characterized by their ability to execute instructions stored in a memory, while the passive modules (the memory modules) can only respond to externally applied commands.

Each major module has 16 ports (numbered 0-15) which are used to connect active modules to passive modules (An interconnection between an active and a passive module uses a single port on each.) Each active module is connected to each passive module. Each such interconnection is a single instance of the "major module interface".

Active modules operate independently of each other and may communicate only with passive modules. An active module requests access to a specific passive module, and this request is granted asynchronously according to a prewired priority scheme. Communication between active modules is accomplished by first passing the message to a specific passive module, which then forwards the message to the second active module.

The hardware also includes two types of I/O devices: magnetic recording tape handlers and typewriter terminals. Each such device is directly connected to a specific I/O controller (perhaps via a switching network). The connection is called the "tape handler device interface" or the"typewriter terminal device interface".

## Identification

Major module interface

## Purpose

The major module interface provides the standard interconnection for all active and passive modules in the hardware system. By this interface a port on each active module (i.e. each processor or I/O controller) is connected to a port on each passive module (i.e., each memory module) in the system. Thus, in a system with m active modules and n passive modules there would be m times n instances of the major module interface.

## Introduction

Each instance of the major module interface consists of 112 lines, arranged in 7 groups. The following table lists these groups, the number of lines in each, and the direction of communication.

| group name | number of lines | direction of communication |
|---|---|---|
| data | 64 | either |
| memory address | 40 | active to passive |
| command | 3 | active to passive |
| access request | 1 | active to passive |
| operation completed | 1 | passive to active |
| final status | 2 | passive to active |
| interrupt present | 1 | passive to active |

## Data lines

The 64 data lines communicate a single word of information in either direction. If the command implies data movement from active to passive, such as it would for an operation storing a word in memory, then the data lines carry the information in that direction. If movement in the other direction is implied, such as for an operation reading a word from memory, then the data lines carry the word in the opposite direction. For any given command, however, communication via these lines occurs in one direction only.

## Memory address lines

The 40 memory address lines carry from the active module to the passive module the absolute address of the memory location to be referenced.

## Command lines

The 3 command lines carry from the active module to the passive module the code indicating the specific operation being requested of the passive module. The following table lists the legal commands, and indicates the direction of information flow over the data lines for each command.

| command | mnemonic | code | data communication |
|---|---|---|---|
| read | READ | 001 | passive to active |
| write | WRITE | 010 | active to passive |
| read and hold | RAH | 011 | passive to active |
| set interrupt cell | SIC | 100 | active to passive |
| execute interrupt | EXI | 101 | passive to active |

All unassigned codes result in the response "illegal action". (See below, final status lines.) The section of the notebook describing the memory modules indicates the specific response to each of these commands.

## Access request line

Once an active module has set the data, address, and command lines with the signals specifying the operation desired, a signal is placed on the access request line to indicate to the connected passive module that service is desired. The signal remains until the operation is completed and the results noted by the active module.

## Operation completed line

When the requested operation is completed, the passive module places a signal on the operation completed line to inform the initiating active module that all output lines are set.

## Final status lines

The 2 final status lines return to the active module the termination state of the operation. The following table lists the possible states and the associated codes.

| state | code |
|---|---|
| operation successfully completed | 00 |
| non-existent address | 01 |
| parity error | 10 |
| illegal operation was requested | 11 |

## Interrupt present line

The interrupt present line is a manifestation of the inter module control communication facility of the hardware system. A passive module sets a signal on this line to its control module when the interrupt cell in the passive module is set by an active module. See the notebook section on control communication for further description and references.

## Communication

A typical use of the major module interface originates in an active module. Consider the example of writing a word into memory. Once the data is generated by a processor and the absolute address in which it is to be stored is computed, the port selection logic of the processor determines which of the 16 processor ports is connected to the memory module containing the desired address. The data and the address are set on the data and address lines, and the code for the command WRITE is set on the command lines. A signal is then set on the access request line.

The memory module connected to the selected port will complete service on requests from all lower numbered memory module ports, and then recognize the signal from this active module on the access request line. The indicated command is then performed.

When the storing operation is completed by the memory module, the final status lines are set by the memory module to indicate the outcome (assuming everything worked, they are set to 00) and a signal is set on the operation completed line. The final status and operation completed line signals remain until the access request signal from the processor initiating the action is removed, indicating that the output has been received and processed.

It is also possible for passive modules to initiate communication with active modules at the request of other active modules. Such is the case when the interrupt cell in a passive module is set in response to an SIC command to the passive module

Date: DEC 16 1968

## Identification

Control communication

## Purpose

The active modules of the hardware system communicate with each other through the standard major module interface to the passive modules. The passive modules act as messengers, passing along communications received to the proper active module. This section describes the overall scheme of interrupt communication. The specific details, as they apply to the three kinds of modules involved, are given in the sections for those modules.

## Introduction

As previously described, each active module is connected to each passive module through the standard major module interface. The connections are to 16 ports on each module. One of these ports on each passive module is designated by switch setting as the control port for that memory. The processor or I/O controller connected to the control port is the control module associated with the passive module. In a normal system configuration each active module is the control module for at least one memory module, and there are at least as many passive as active modules.

When a passive module receives a control communication, called an interrupt, from any of the up to 16 active modules connected to it, it always forwards this message to its control module. Thus, in order for one active module (A) to send an interrupt to another (B), the passive module (C) controlled by B must be known by the number of the port on A connecting A to C.

## Interrupts

Contained in each passive module (in the system controller portion of the memory module) is an interrupt cell. This cell may be set by any active module by sending the command SIC to the memory module. When the cell is set the interrupt present line to the control module for the memory module is set with the signal. When next convenient, the control module will recognize the signal on this line, and request access to the passive module, indicating the command EXI (execute interrupt). The passive module will then respond by setting the cell "off" and removing the signal from the interrupt present line. The response of the active module to executed interrupts depends on whether the active module is a processor or an I/O controller. Processors make an unconditional CALL to the location specified by a special program loadable register. I/O controllers retrieve an instruction word from a pre-specified memory location. See the descriptions of each of these modules for the details of their reaction.

## Identification

Tape device interface

## Purpose

The tape device interface is used to connect each magnetic recording tape handler to a specific I/O controller. The details of this interface are not contained in this version of the Clics notebook.

## Identification

Typewriter terminal device interface

## Purpose

The typewriter terminal device interface is used to connect each typewriter terminal directly to a specific IOC, or to connect output taps from a switching network directly to a specific IOC. In the latter case, the typewriter terminals are attached to the input taps of the switching network. The details of this interface are not contained in this version of the Clics notebook.

Date: DEC 16 1968

## Identification

Hardware configuration

## Purpose

In this section the hardware configuration of the current Clics system is specified. This configuration is not maximal, allowing for future expansion if circumstances warrant such growth.

## Specification of modules

The current configuration includes the following modules and devices:

4 processors
2 input/output controllers
12 memory modules, two of which contain clocks
12 tape handlers
100 typewriter terminal lines from typewriter terminal switching network

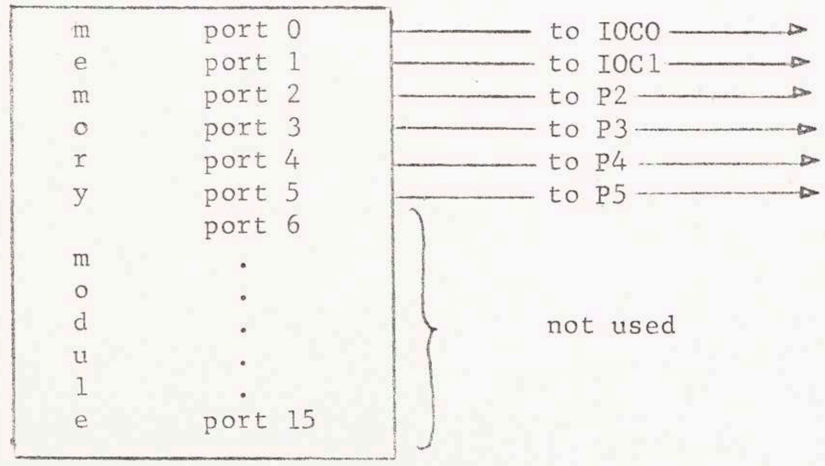
The processors are labelled P2, P3, P4, and P5. The input/output controllers are labelled IOC0 and IOC1. The memory modules are labelled M0 through M11.

Each input/output controller contains 6 data channels for tape handlers (channels 3, 5, 7, 9, 11, and 13) and 50 data channels for typewriter terminals (channels 15, 17, ... 113).

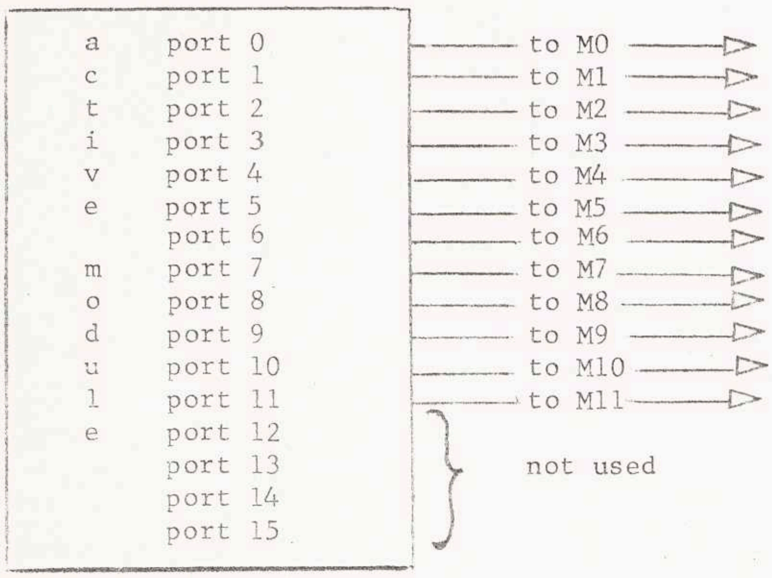
Memory modules M0 through M11 contain in consecutive order locations 0 through 805,306,367. Each memory module contains an equal amount of storage.

## Port assignments

The following diagram indicates the port assignments for each of the 12 memory modules.



The port assignments for the active modules are indicated by the following diagram:



## Control relationships

IOC0, IOC1, P2, P3, P4, and P5 are the control modules for M0, M1, M2, M3, M4, and M5, respectively. Memory modules M6 through M11 do not require control modules.

## Clocks

The primary system clock is contained in memory module M4. The backup system clock is contained in memory module M5. The following table indicates the address by which each may be referenced.

| clock | address (in decimal) |
|---|---|
| primary calendar | 268,435,456 |
| primary alarm | 268,435,457 |
| secondary calendar | 335,544,320 |
| secondary alarm | 335,544,321 |

The locations specified are the first two in M4 and M5.

## Interrupt communication

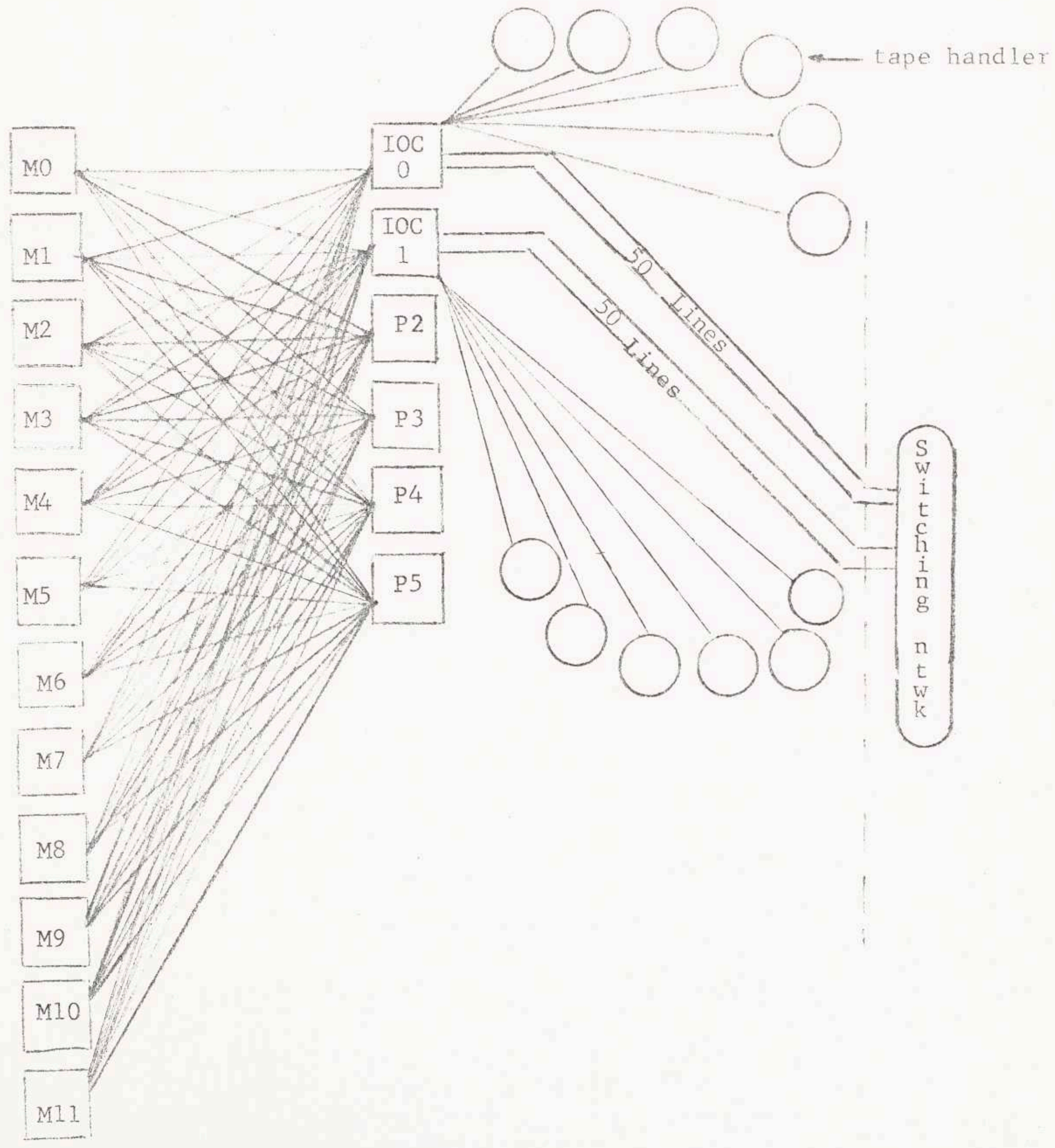
The interrupt cells in memory modules M0 and M1 (the memory modules controlled by the two input/output controllers) are used to forward control communication to these active modules. The setting of the interrupt cell in M0, for example, causes IOC0 to retrieve the contents of its control channel mailbox. See the description of the input/output controllers for further details on control communication to initiate input/output.

Memory modules M2 and M3 are designated by switch setting in the input/output controllers as the recipients of the SIC commands from IOC0 and IOC1, respectively. Thus, P2 and P3 receive the _interrupt present_ line signals generated.

Since M4 and M5 contain the clocks, it is the interrupt cells in these modules that are set when the alarm "rings", and P4 and P5 receive the _interrupt present_ line signals generated.

## Diagram

Following is a diagram illustrating the hardware configuration.



Clics System Configuration

## Identification

Overview of processor capabilities

## Purpose

The processors provide the computation power of the Clics system. Processor capabilities are one of the major resources provided by Clics to its community of users. This section describes these capabilities in general terms.

## Introduction

The hardware and the control programs of the Clics system are highly functionally dependent. This dependency is perhaps most obvious in the structure of the Clics processors, for they contain logic which can function properly only in the "software" environment provided by the control programs of the Clics system. In particular, the processors are designed to execute within the segmented address space of a Clics process, and contain logic for addressing and for enforcing access constraints within such an address space. Because the processors are designed specifically for the Clics system they are able to perform much of the work assigned to the control programs in systems with less well suited processors. While this complicates the processors themselves, it reduces the complexity of the control programs by more than an equal amount.

The capabilities of a Clics processor may be grouped into the categories of manipulation and decision, access, protection, and communication. The first is the general purpose computation ability of the processor, and is manifest in the instruction set and programmable registers. Within the second lies the segment addressing logic while the third is embodied in the procedure ring register and the access mode indicator in the segment descriptor words. The final category is found in the ability of each processor to send and receive interrupts to and from other processors (and the Input/Output controllers) in the system.

We will now briefly describe each of these categories.

## Manipulation and decision

The general purpose aspects of each processor are representative of many large scale general purpose computers. The instruction set contains approximately 25 operations to perform logical manipulations and integer (binary) arithmetic. While this instruction set is relatively small, each operation may be suffixed with a large number of operand specification types, effectively increasing the number of instructions. The 16 general purpose registers of 64 bits may each function as pointers, index registers, and arithmetic or logical registers. These, for the most part, are completely programmable, although a few are reserved by the processor or by control program convention for special purposes.

## Access

The access capabilities of each Clics processor directly reflect the structure of the segmented address space of a Clics process. Each processor contains logic to map addresses given as segment number and word number into the absolute address required for memory references. The address space of the process executing on a processor is manifest in the descriptor base register, a special processor register which contains the absolute address of the base of the descriptor segment for the process. Because this register is the only processor manifestation of the address space, the processor can be quickly switched to operate in the address space of another process. Special instructions are provided to load and store the descriptor base register. Addresses contained in instruction words and indirect words are normally given as a segment number and a word number, with absolute address specification used only in a few control program procedures, and at system load time.

## Protection

The protection mechanisms in the Clics processors are designed to implement the concept of rings of protection within each process. The procedure ring register contains the ring number (0 through 7) of the ring in which the procedure segment that is executing resides. This ring number is compared with the access mode indicator of each segment referenced by the procedure. If the attempted access is not within the access constraints indicated by the access mode indicator a fault is caused and control is taken from the erring procedure. It is the characteristic of rings that procedures residing in lower numbered rings have greater access privileges than those in higher numbered rings. The ring with the most privileges, ring 0, is reserved for system-wide control programs. Several instructions with the power to alter the current system status are executable only when the processor is in ring 0 i.e., only when the procedure ring register contains zero. The processor's ability to fault when an error occurs or an illegal specification is encountered is also included under protection. A fault forces control to a pre-specified location.

## Communication

Each processor has the capability to send interrupts to other active modules and to accept them. An interrupt, when received, causes the processor to stop executing its current instruction sequence and transfer control to an instruction whose location is specified by one of the reserved general purpose registers. Complete information about the state of the processor when the interrupt was recognized is automatically saved so that the interrupted instruction sequence can be resumed later, if appropriate. The specific response to each interrupt is determined by the particular control program invoked by the forced transfer of control.

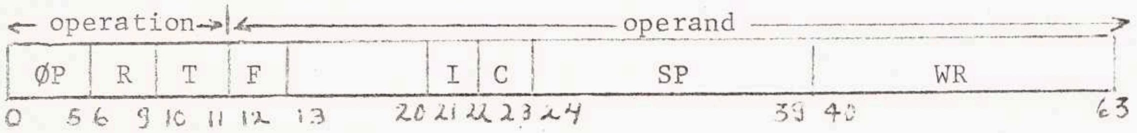
## Identification

Processor dependent storage formats

## Purpose

Words retrieved from memory as instruction words, indirect words, segment descriptor words, page map words, or binary numbers are given specific interpretations by a processor. This section describes the format expected by the processors for each of these.

## Instruction and indirect words



Instruction words are divided into two parts: the operation specification and the operand specification. The operation specification specifies the machine instruction to be executed, the general purpose register to be manipulated (if any) and the type of the operand specification. If the type is memory then the operand of the instruction is the memory word whose address is indicated by the operand specification. If the type is register then the operand is a general purpose register whose number is indicated by the operand specification. If the type is immediate then the operand is the operand specification itself, extended appropriately to 64 bits. In each case the operand specification also includes a fault bit which if "on" when the instruction is executed will cause an instruction word fault.

Indirect words have the same format as instruction words. The operation specification, however, is ignored as the operation of indirection is implied by the word being referenced during an indirect cycle. The operand specification is assumed to be of type memory, and thus indicate an address of a word in memory. The fault bit, if "on" when the indirect word is referenced, will cause an indirect word fault.

The following table specifies the use of the fields of the operation specification portion of an instruction word.

| Field | Use |
|---|---|
| OP | Operation code |
| R | Number of general purpose registers directly manipulated (if any) |
| T | Type of operand specification<br>00 - not allowed, caused an illegal procedure fault<br>01 - memory<br>10 - register<br>11 - immediate |

If the type is memory, then the operand specification indicates directly or indirectly the address of the operand in memory. The operand specification of an indirect word is also interpreted in the same way as indicating an address in memory. The following table specifies the use of the fields.

| Field | Use |
|---|---|
| F | Fault bit |
| I | Indirect bit. When "on" the remaining fields address an indirect word rather than the operand. |
| C | Address interpretation control.<br>00 - SP and WR specify a (segment number\|word number) address.<br>01 - SP specifies in its last four bits the number of a general purpose register to be used as a pointer. WR specifies the offset relative to the (segment number\|word number) address in that register.<br>10 - A (segment number\|word number) address is constructed from the S field of the general purpose register specified by SP, and the WR field of the instruction.<br>11 - SP and WR are concatenated to form an absolute address. |
| SP | Depending upon C, contains a segment number, a register number, or is part of an absolute address. |
| WR | Word number, offset, or part of absolute address, depending upon C. |

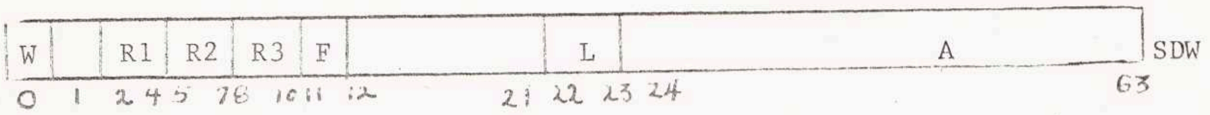
If the operand specification of an instruction is register then the last 4 bits of the WR field are interpreted as the number ~~of the~~ general purpose register that is the operand.

If the operand specification of an instruction is immediate then the last 51 bits of the instruction word, padded on the left to 64 bits, are the operand. (The two types of padding used are described elsewhere.)

In both of the last two cases F is interpreted as a fault bit, causing an instruction word fault if discovered "on".

## Segment descriptor words

The words of a descriptor segment are called segment descriptor words. Each may contain the absolute address, and the access mode indicator of a segment in the address space of the process defined by the descriptor segment.



| field | use | |
|---|---|---|
| W | Write permission bit. | access mode indicator |
| R1 | Ring indicator 1. (Top of write bracket) | |
| R2 | Ring indicator 2. (Top of read and execute bracket) | |
| R3 | Ring indicator 3. (Top of call bracket) | |
| F | Fault bit. If this bit is one when this SDW is referenced a segment descriptor fault will occur. | |
| L | Number of levels of page maps for the corresponding segment.<br>00 - Segment has zero levels of page maps (none). A is the absolute address of the beginning of the actual segment.<br>01 - Segment has a single level of page maps. A is the absolute address of the page map.<br>10 - Segment has two levels of page maps. A is the absolute address of the master page map.<br>11 - Not allowed - causes illegal procedure fault. | |
| A | Absolute address of the base of the segment corresponding to the SDW. | |

## Page map words

A page map word consists of the F and A fields of a segment descriptor word, in the same positions in the 64 bit word. A contains the absolute address of a page of the segment.

## Representation of binary numbers

Binary numbers are represented in standard two's complement form. Bit 0 of a word is the sign bit, with zero indicating plus and one indicating minus.

Date: DEC 16 1968

## Identification

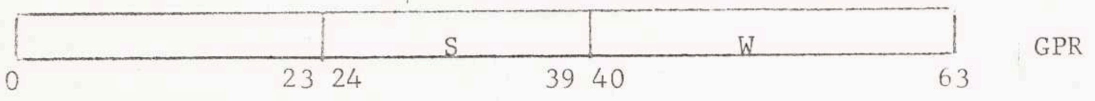
Program accessible registers

## Purpose

The program accessible registers are those that may be directly referenced by instructions. This section describes the format and use of these registers.

## General purpose registers

The 16 general purpose registers may be used for general arithmetic or logical functions; or may function as pointer registers. A pointer register contains a segment number and a word number. An instruction word or indirect word may address relative to the pointer by having an address interpretation control field with the proper value. The offset in WR is then added to the word number in the indicated pointer register to form a (segment number | word number) address.



The S and W fields contain the segment number and word number if a GPR is being interpreted as a pointer register. These field designators are ignored when a GPR is used in other manners.

GPR0 through GPR2 are reserved by the processor for special functions. GPR3 through GPR5 are reserved by control program convention as special pointer registers. The following table indicates these assignments.

| GPR | use |
|---|---|
| 0 | instruction pointer (ip): points to the instruction being executed by the processor. |
| 1 | temporary pointer (tp): points to the instruction word or indirect word being evaluated by the processor. |
| 2 | call pointer (cp): points to the entry to be called in the event of an interrupt or a fault. |
| 3 | stack pointer (sp): points to the current frame in the procedure stack. |
| 4 | linkage pointer (lp): points to the beginning of the current linkage segment. |
| 5 | argument pointer (ap): points to the head of an argument list on a subroutine call. |

## Indicator register

The indicator register reflects processor conditions resulting from instruction execution, and from external interrupts

```
|Z|N|C|Ø|I|F|   IR
 0 1 2 3 4 5 8
```

| field | use |
|---|---|
| Z | zero indicator |
| N | negative indicator |
| C | carry indicator |
| Ø | overflow indicator |
| I | Interrupt indicator |
| F | Fault code |

The zero, negative, carry, and overflow indicators are set by the execution of instructions. The description of each instruction specifically indicates which indicators are affected. The "I" and "F" fields are set when faults or interrupts occur. If a fault has occured, "I" is set to 0 to indicate the fault, and "F" is set with the code identifying the fault. See the section describing the fault mechanism for the fault names associated with each code. If an interrupt occurs, "I" is set to 1, and "F" is ignored.

## Procedure ring register

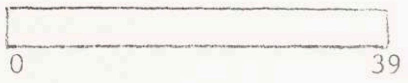
The procedure ring register specifies by number (0 - 7) the ring in which a processor is executing.

```
|     |   PRR
 0   2
```

The procedure ring register may be program altered only to a value greater than it initially contained. Only the CALLn instruction, or a fault or interrupt may alter it smaller.

## Descriptor base register

The descriptor base register contains the absolute address of the page map for the descriptor segment of the process executing on the processor.

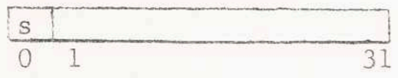


DBR

The descriptor segment is assumed to have a single page map, i.e. one level of page map.

## Interval timer register

This register may be program loaded with a signed binary number. Each time the processor references memory it is decremented by one. When it becomes negative a timer runout fault is generated. (Generation occurs at the beginning of the next instruction.) The fault will not occur when the processor is operating in ring 0. The timer, however, continues to be decremented, and can in such cases become negative.



ITR

## Interrupt inhibit switch

The interrupt inhibit switch inhibits recognition of the interrupt present signal from the controlled memory module. It may be set "on" and "off" in ring zero by a special instruction. If it is "on" when control passes out of ring zero it is automatically reset.

## Identification

Operand specification interpretation.

## Purpose

The operand specification portion of an instruction word may be three types: memory, register, and immediate. This section describes the interpretation of each of these types.

## Introduction

The operand of an instruction may be a memory word, a register, or the operand specification fields of the instruction itself. The operand of an indirect word is always a memory word. In each case the operand specification identifies the operand. Following is a description of the interpretation of that specification in each of the three cases.

## Memory

Operand specifications of type memory (T = "01"b) are perhaps most common. This type means that the I , C , SP , and WR fields of the instruction directly or indirectly address the operand of the instruction in memory. The address may be directly indicated in four ways. (If the "I" bit is off, then the address is direct.) An example for each way will be given. The C field of the instruction word differentiates the ways. (Note: the contents of instruction word fields are represented in decimal, unless otherwise indicated.)

example: C = "00"b

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ADD | 7 | 1 | 0 | | 0 | 0 | 17 | 4267 |
| OP | R | T | F | | I | C | SP | WR |

C = "00"b means that the instruction word contains the (segment number|word number) address of the operand in the SP and WR fields. In this case, the contents of word 4267 of segment 17 would be added to the contents of GPR7.

example: C = "01"b

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ADD | 7 | 1 | 0 | | 0 | 1 | 0 | 417 |
| OP | R | T | F | | I | C | SP | WR |

C = "01"b means that the word number given is relative to the pointer register (GPR) whose number is given by the last four bits of SP. The (segment number|word number) address of the operand is formed by taking the segment number from the S field of the indicated GPR, and constructing a word number by adding the W field of the GPR to the WR field of the instruction word. In the example given, since the GPR given is the instruction pointer, the contents of the word that is 417 location beyond this instruction word in the procedure segment would be added to the contents of GPR7.

example: C = "10"b

| OP | R | T | F | | I | C | SP | WR |
|---|---|---|---|---|---|---|---|---|
| ADD | 7 | 1 | 0 | | 0 | 2 | 0 | 417 |

C = "10"b means that the SP field contains the number of a GPR in its last four bits. The (segment number|word number) address of the operand is the segment number from the S field of this GPR, and the word number from the WR field of the instruction word. The W field of the GPR is ignored. In the example given, since GPR0 is the instruction pointer, the contents of word 417 of the containing procedure segment would be added to the contents of GPR7.

example: C = "11"b

| OP | R | T | F | | I | C | SP | WR |
|---|---|---|---|---|---|---|---|---|
| ADD | 7 | 1 | 0 | | 0 | 3 | 72 | 2174 |

C = "11"b means that the SP and WR fields are to be concatenated to form an absolute address. In the example the contents of word ($72*2^{22}$ + 2174) would be added to the contents of GPR7. Absolute address specification may only be used in ring 0.

If the "I" bit is "on" in an instruction, then the operand is addressed indirectly. The address computed in any of the four ways described above is not that of the operand, but that of an indirect word. The indirect word is retrieved from memory, and its operand specification portion evaluated exactly as described above. The indirect word may indicate further indirection. The address of the operand of the instruction is specified by the first indirect word encountered that does not have the indirect bit"on." The access privileges of an indirect word are consistent with the highest ring in which the indirect word may written. Thus, if indirection is indicated through an indirect word that may be written from a higher ring than that in which execution is occuring, reference may only be made to words accessible from the higher ring.

## Register

Operand specifications of type register (T = "10"b) are allowed only with group 1 instructions. This type means that the last four bits of the WR field is interpreted as the number of a GPR. The operand is this register.

## Immediate

Operand specifications of type immediate (T = "11"b) are also allowed only with group 1 instructions. In this case the operand of the instruction is bits 13-63 of the instruction word itself, extended to 64 bits by padding to the left. If the instruction is arithmetic, padding is done with copies of bit 13 of the instruction word, the sign bit of the immediate operand. If the instruction is logical, padding is done with binary zeros.

Date: DEC 16 1968

## Identification

Address mapping

## Purpose

This section describes the transformation of a (segment number | word number) address to an absolute address. Addresses are normally specified within a process as a segment number and a word number. The segment number identifies a single segment within the process's address space, and the word number indicates the offset within that segment. The processor, however, must retrieve words from memory by absolute address. Thus, each memory address generated by a program must be converted from its (segment number,| word number) form to an absolute form by the processor. This transformation is accomplished by the address mapping logic of the processor.

## Introduction

In order to understand the method of the transformation, it is necessary to understand something of the environment in which it takes place, the address space of a Clics process. Such an address space consists of many procedure segments and data segments. Each segment is defined and included in the address space by directly calling the storage management subsystem, and is described by a segment descriptor word (SDW) in the process's descriptor segment. A segment's number is the word number of the corresponding SDW in the descriptor segment. A SDW contains, among other things, an absolute address and a level number. The absolute address is the location in memory of the base of the corresponding segment, and the level number indicates the number of levels of page maps interposed between the SDW and each word of the segment. The absolute address of the base of the descriptor segment is kept in the descriptor base register of the processor. The memory space occupied by the segments in a process is allocated in individual 256 word blocks.

## Address mapping

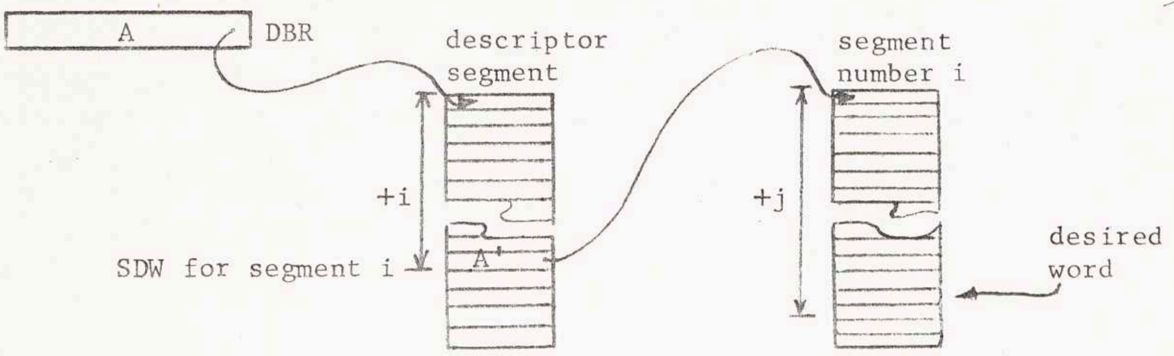
The formation of an absolute address may now be considered. Assume for the moment that the descriptor segment and segment number "i" each occupy a single block of memory. The absolute address of the SDW for segment number "i" is then computed as

$$A + i$$

where "A" is the absolute address in the DBR. The absolute address of word number "j" from segment number "i" would then be

$$A' + j$$

where "A'" is the address contained in the SDW whose absolute address was computed in the previous step. The following diagram summarizes these operations.



The above transformation works very well if both the descriptor segment and segment number "i" are a block or less in length, for this gurantees that each will occupy a contiguously located group of words. However, consider the case in which segment "i" contains more than the 256 words that are in a single block of memory. Since a block allocation scheme is used to assign memory to each segment, there is no gurantee that the blocks of this segment will be contiguously located. The multiple blocks occupied by a segment are called "pages", and a page map is provided to bind the pages together into a segment. Each page map word (PMW) contains the absolute address of a single page of the segment. Since the page map is also contained in a memory block, there can be, at most, 256 PMW's in a page map. The PMW's are ordered so that word "k" of a page map describes page "k" of the segment, which contains words "k*256"through "(K+1)*256-1" of the segment. In the case of a segment with a page map, the SDW contains the absolute address of the page map. The level number in the SDW is used to distinguish between segments having 0 levels of page map (the previous example) and those having 1 level of page maps. Segments with 0 levels are necessarily restricted to less than 257 words. Segments with 2 levels of page tables are also possible, and are described later.

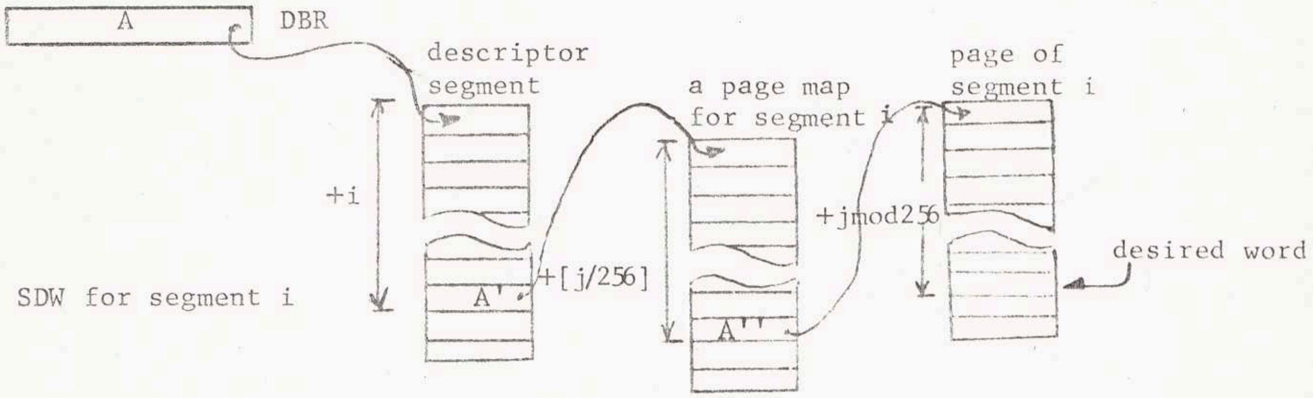
Once the SDW for segment number "i" has been located, computing the address of word number "j" now involves some additional steps. The level number in the SDW is checked. If it is 0, then the address is computed as described above. If it is 1, then the address of the proper PMW must first be computed as

$$A' + [j/256]$$

where"A'" is defined as above, and [ ] indicates the"integer part of." The absolute address of word number "j" is then computed as

$$A'' + (j \text{ modulo } 256)$$

where "A''" is the address contained in the PMW whose address was computed in the previous step. The following diagram summarizes this example.



In order to allow segments larger than $256^2$ words, multiple page maps are used. In these cases the level number in the SDW is 2, indicating two levels of page maps. The SDW contains the absolute address of what may be called the master page map for the segment. The PMW's of this master page map each contains the absolute address of another page map, which points to the actual pages of the segment. Computation of the absolute address of a word in such a segment is done by a direct extension of the method for segments with one level of page maps. A maximal sized segment with 2 levels of page maps contains $256^3$ words. Since $256^3 = 2^{24}$, and there are 24 bits in all hardware word number fields, 2 levels of page maps are the most ever required.

The processors enforce the convention that the descriptor segment always has a single level of page map. Thus, $256^2$ SDW's are possible. Since $256^2 = 2^{16}$, and there are 16 bits in all hardware segment number fields, one level is enough. Finding the SDW for a segment number involves two steps, first finding the proper PMW, and then locating the SDW.

## Identification

Segment access protection

## Purpose

The extent to which instructions in the executing procedure of a process may direct the processor to read, write, execute, or call words in the segments of the process is determined by the relationship of the protection ring in which the processor is operating to the access mode indicators of the segments. The processor automatically determines that the proper relationship exists before performing any such requests. Detection of an improper relationship produces an illegal procedure fault which removes control from the erring procedure. This section describes the action of the process in performing such validation in general terms. Details of the specific hardware mechanisms appear in the various notebook sections describing processor logic units.

## Introduction

Each processor includes a procedure ring register (PRR) which defines the number of the protection ring in which the processor is operating. The three bits of this register can contain the binary ring number '000' through '111'. Abstractly, a protection ring is a degree of ability to access the words in the segments of the process. From the manner of implementation ring zero represents the greatest degree and ring seven the least.

Associated with each segment in the address space is an access mode indicator. It occupies the first eleven bits of the corresponding segment descriptor word and contains a write permit bit and three ring numbers. Its format is:

```
| W |   |     R1     |    R2    |    R3    |
0   1  2   3   4 5 6    7 8  9   10          bit
```

The processor will perform a request to read a word from a segment only if PRR $\leq$ R2 , where R2 is from the segment's access mode indicator. It will perform a request to write the word only if PRR $\leq$ R1 and W = 1. It will allow the locus of control to be transfered to a word via a call only if PRR $\leq$ R3 . R1, then, defines the write bracket of the segment; R2 the read bracket; and R3 the call bracket. The bottom of each bracket is the lowest ring number, zero.

From the above inequalities it is clear that protection rings are inclusive. Instructions in ring "i" are allowed access to all segments that instructions in ring "i + 1" may reference. In addition, they may write, read, or call those whose R1, R2 and R3 fields contain "i", where instructions from "i + 1" may not.

Ring zero is reserved by system convention for the execution of system procedures. System-wide data bases which must be protected against casual access by user procedures occupy segments whose R1 and R2 ring number are zero. Thus, only the system procedures may reference them. Such procedure and data segments are informally referred to as "being in" or "occupying" ring zero. Because in normal use all segments of a process tend to have access mode indicators in which R1, R2 and R3 are equal, they are also referred to as "being in" the ring so specified.

User provided procedures operate in rings other than zero. A user initially gains control of a process operating in ring four. Control may be passed to procedures in lower rings only through the use of the special CALLn instruction of the processor. Passing control to a higher ring is accomplished by resetting the PRR.
Recognizing the special meaning of ring zero certain processor instructions which have the power to alter the status of the system, thus affecting other processes, will be executed by the processors only when they are operating in ring zero. Likewise, no procedure may cause the contents of the PRR to be altered downward, other than with the special CALLn instruction, thus altering its access privileges.

## Instruction fetch

To see how the ring mechanism works consider the execution of an instruction by a processor. Initially the instruction pointer (GPR0) contains the (segment number|word number) address of the instruction and PRR contains the number of the protection ring in which the processor is operating. The processor must first retrieve the instruction. Since the instruction pointer was supplied by the execution of the previous instruction the processor may not assume that it addresses an instruction word in a segment that may be read from the current ring. Thus, it first compares PRR and the R2 field from the access mode indicator of the containing segment. If $PRR \leq R2$ is true the instruction may be retrieved. The number in R2, then, also defines the highest ring in which instructions in a procedure segment may be retrieved for execution by a processor, and thus defines the execute bracket of the segment. Of course, if the condition is false an illegal procedure fault results.

## Effective address formation

Once the instruction is retrieved its operand specification portion must be evaluated. If the specification type is register or immediate no further access validation is required as execution of the instruction will not reference other words in segments. In the case where its type is memory, indicating that the operand is a memory word, further validation is required. The specification is a hardware representation of an address expression. Transformation of it to the actual (segment number|word number) address of the operand is termed effective address formation.

The contents of the PRR are used to initialize a temporary ring register (TRR). If the effective address formation can be done without reference to more words in memory (indirect words) then TRR contains the ring number against which to validate the final operand access. If indirection is involved, however, things are more complex, for indirect words are retrieved by reading them - which involves validation. Before each indirect word is read, the condition TRR $\leq$ R2 , where R2 is from the access mode indicator of the segment containing the indirect word, must be checked and found to be true. If true the indirect word is retrieved and evaluated.

It is entirely possible, however, that the indirect word is from a segment that may be written from a higher ring than that currently indicates by PRR. Thus the indirect word may have been set by a procedure operating in a higher ring - a procedure with less access privileges. In fact, that procedure, if untrustworthy, could violate the protection mechanism and force this instruction to manipulate the wrong word. For example, a procedure in ring seven might call a system procedure in ring zero that in the normal course of its action stored a value back into an argument accessed via an indirect word supplied by ring seven. The caller might set that indirect word to point to a vital location in a ring zero data base, a location it could not access itself, and then let the ring zero procedure unknowingly store into it by mistake, adversely effecting the entire system.

To prevent this, indirect words are allowed access privileges that correspond only to the highest R1 value encountered in an indirect chain so far. After each indirect word retrieval,the R1 value associated with the containing segment is compared with TRR . If TRR $<$ R1 then R1 replaces TRR . The next indirect word retrieval follows the same pattern, with validation relative to TRR .

When the indirect word chain is completed, and the effective address of the operand is determined, TRR will contain the highest R1 value encountered.

## Operand reference

Assuming successful effective address formation, the processor is now ready to perform the instruction. This may involve reading, writing or not referencing the operand. The TRR value is used to validate the read or write reference. A read is performed if TRR $\leq$ R2 is true, and a write if TRR $\leq$ R1 and W = 1 is true, where W, R1, and R2 are from the access mode indicator of the operand containing segment. If the operand is not referenced then the instruction is either EAPn or CALLn, both special cases.

## EAP instruction

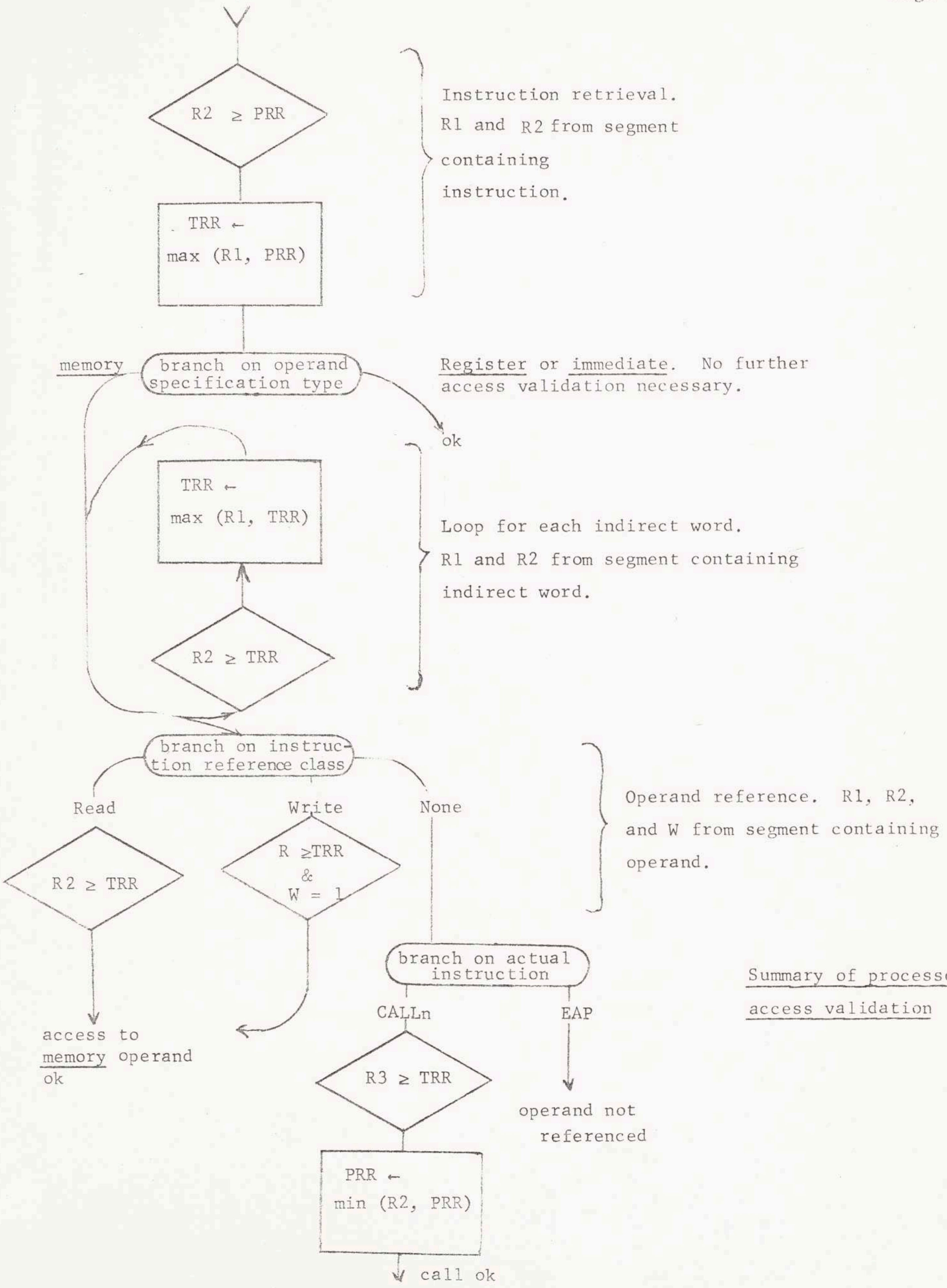
The EAP instruction causes the formed effective address to be placed in a GPR. The memory word addressed is not referenced, and thus no further access validation is necessary.

## CALL instruction

The CALL instruction is used to reset the instruction pointer and save the current machine registers, thus transfering the locus of control in such a way that a return can be made. In the most general case the operand of a CALL may be an instruction which can only execute in a lower ring. (That is, R2 for the new procedure segment may contain a value less than PRR .) It is also the case that some procedure segments should not be CALLable from all rings. A CALL is performed only if TRR $\leq$ R3 is true, where R3 is from the access mode indicator of the CALLed procedure segment. If the CALL is validated then PRR is reset with the smaller of PRR and R2 . If the CALL comes from above R2 of the CALLed segment then the ring of the processor changes to that specified by R2. If it comes from below then the ring remains the same. The value of the PRR before the CALL is saved in the procedure stack of the ring CALLed, and replaced upon return.

## Summary

The flow diagram on the following page summarizes the access validation described above. Conditions expressed in diamond shaped boxes must be true for execution to continue. If any false conditions are encountered, an illegal procedure fault results.


R2 ≥ PRR
TRR ← max (R1, PRR)
Instruction retrieval. R1 and R2 from segment containing instruction.
memory
branch on operand specification type
Register or immediate. No further access validation necessary.
ok
TRR ← max (R1, TRR)
Loop for each indirect word. R1 and R2 from segment containing indirect word.
R2 ≥ TRR
branch on instruction reference class
Read
Write
None
Operand reference. R1, R2, and W from segment containing operand.
R2 ≥ TRR
R ≥TRR & W = 1
branch on actual instruction
CALLn
EAP
access to memory operand ok
R3 ≥ TRR
operand not referenced
PRR ← min (R2, PRR)
call ok

<u>Summary of processor access validation</u>

## Identification

Processor instruction set

## Purpose

This section describes the instruction set of a Clics processor.

## Introduction

The instruction set of a Clics processor is divided into two groups according to the types of operand specification accepted. The groups are: instructions which accept memory, register, and immediate operand specifications; and those which accept only memory operand specifications. The groups are labeled 1 and 2 respectively.

## Group 1 instructions

Instructions in group 1 have the common property that they read their operand. Logical instructions in group 1 use logical extension of immediate operands, and arithmetic instructions use arithmetic extension. The group 1 instructions are:

logical -

| | |
|---|---|
| ANDn | AND to register n |
| CMPLn | Compare logical to register n |
| EØRn | EXCLUSIVE OR to register n |
| LIT | Load interval timer |
| LOADn | Load register n |
| ORn | OR to register n |
| RLSn | Register n left shift |
| RRSn | Register n right shift |
| SIC | Set interrupt cell |
| SIIS | Set interrupt inhibit switch |
| TIR | Test indicator register |

arithmetic -

| | |
|---|---|
| ADDn | Add to register n |
| CMPAn | Compare arithmetic to register n |
| DIVn | Divide register n |
| MPYn | Multiply register n |
| SUBn | Subtract from register n |

## Group 2 instructions

Group 2 instructions require operands of type memory, and may read, write, or not reference their operands. The group 2 instructions are:

| | |
|---|---|
| ADDSn | Add register n to store |
| CALLn | Call, saving through register n |
| EAPn | Effective address to pointer n |
| LDBR | Load descriptor base register |
| RMCn | Restore machine conditions through register n |
| SDBR | Store descriptor base register |
| SIT | Store interval timer |
| STOZn | Store register n if operand zero |
| STOn | Store register n |

## Operation code assignments

Operation codes are assigned to facilitate separation of instructions into groups. The following general assignment rules are used:

1. Ø0 through Ø17 are group 1 - logical.

2. Ø20 through Ø37 are group 1 - arithmetic.

3. Ø40 through Ø57 are group 2.

4. Ø60 through Ø77 are not currently used.

## Instruction description format

Each instruction is described in a standard format that contains eight sections. Below is a sample description.

Mnemonic: The standard mnemonic for the instruction.
Title: The title of the instruction.
Operation code: The octal operation code for the instruction.
Class: The reference class of the instruction. Possible classes are read, write, and none. The class specifies the kind of reference made to the instruction's operand and reflects the access permission necessary in the containing segment if the operand is a memory word. Of course, if the operand is a register, or is immediate, then access permission is irrelevant. Instructions indicating class none do not reference their operands.
Group: Group number of the instruction.
Description: Summary of the action of the instruction.

Indicators affected: List of the indicators in IR affected by the execution of the instruction, and the conditions under which they are set on or off. Unless indicated otherwise the conditions are checked after execution of the instruction has occured.

## Instructions

The instruction descriptions follow in alphabetic order by mnemonic. Several notes apply to the descriptions.

1. Instruction mnemonics ending with the small letter n indicate that the instruction makes use of the R field of the instruction word. n represents its value, and may be the integers 0 through 15 when the instruction is actually written. It is the number of the general purpose register manipulated.

2. C( ) is the "contents of" operator, and may operate on the abbreviation of a register, or the abbreviation Op which stands for the operand of an instruction. Note that elsewhere in this manual this operator is not explicitly used. For example, GPR1 = GPR2 is written to mean C(GPR1) = C(GPR2). In a later version of the notebook this section will be made consistent with that convention.

3. An effective address is that which results from the application of the processor effective address logic to the operand specification of an instruction. Normally, this is the (segment number|word number) address of the memory word that is the operand. Less often it will be the absolute address of that word. (This can occur only in ring 0 when an instruction or indirect word using an absolute address is encountered.) Only operand specifications of type memory are processed by the effective address logic.

4. Instructions whose execution in a specific circumstance will alter the contents of GPR2 are executed in that circumstance only if PRR contains zero, i.e. only if the processor is running in ring zero.

5. Whenever an illegal specification is encountered by a processor in an instruction an illegal procedure fault is generated.

6. The standard register and field abbreviations used in the descriptions are described elsewhere.

Mnemonic: ADDn
Title: Add to register n
Operation code: 20
Class: read
Group: 1 - arithmetic
Description: C(GPRn) is replaced by the sum of C(GPRn) and C(Op)
Indicators affected:
    zero - If C(GPRn) = 0 then on; else off
    negative - If C(GPRn0) = 1 then on; else off
    carry - If a carry out of $GPRn_0$ is generated then on; else off
    overflow - If the range of GPRn is exceeded then on; else off

Mnamonic: ADDSn
Title: Add register n to store
Operation code: 40
Class: write
Group: 2
Description: C(Op) is replaced by the sum of C(GPRn) and C(Op). The operand is read using the RAH memory command. Thus another processor may not reference the operand between reading and writing it.
Indicators affected:
zero - if C(Op) = 0 then on; else off
negative - If $C(Op_0)$ = 1 then on; else off
carry - If a carry out of $Op_0$ is generated then on; else off
overflow - If the range of Op is exceeded then on; else off

---

Mnemonic: ANDn
Title: AND to register n
Operation code: 01
Class: read
Group: 1 - logical
Description: C(GPRn) is replaced by the bit-wise AND of C(GPRn) and C(Op).
Indicators affected:
zero - If C(GPRn) = 0 then on; else off
negative - If $C(GPRn_0)$ = 1 then on; else off

---

Mnemonic: CALLn
Title: CALL, saving through register n
Operation code: 41
Class: None
Group: 2
Description: The effective address of the operand is determined and saved in the hidden CR (call register). If absolute then an illegal procedure fault results. Otherwise, permission to CALL the addressed location is verified and the R2 ring number from the access mode indicatoı of the containing segment determined. The smaller of this number and the C(PRR) is saved in TRR as the ring in which the CALLed location will execute. Next PRR, IR, and GPR0 through GPRn are saved in the n+3 locations at the head of the new procedure stack frame for the procedure being CALLed. The needed (segment number|word number) addresses can be generated by the hardware because by convention the segment number of the appropriate stack segment is the same as the ring number in TRR (the ring in which the CALLed location will execute) and word zero of that stack segment contains an indirect word pointing to the next free frame. Finally, sp (GPR3) is set with the (segment number|word number) address of the new stack frame, PRR is reloaded from TRR, and ip (GPR0) is set with the effective address of the operand from CR. Thus, the next instruction executed by the processor will be the location CALLed.

---

---

Mnamonic: CMPAn
Title: Compare arithmetic to register n
Operation code: 21
Class: read
Group: 1 - arithmetic
Description: The comparison is arithmetic, and negative numbers are considered to be less than positive numbers. If C(Op) = C(GPRn) then the zero indicator is set on, else off. If C(Op) $<$ C(GPRn) then the negative indicator is set on, else off.
Indicators affected: see description

---

Mnemonic: CIMPLn
Title: Compare logical to register n
Operation code: 02
Class: read
Description: The comparison is logical, and the entire 64 bits of the operand and GPRn are treated as a positive binary number. Indicators are set as with CMPAn
Indicators affected: see description

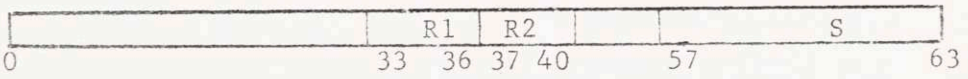
---

Mnemonic: DIVn
Title: Divide register n
Operation code: 22
Class: read
Group: 1 - arithmetic
Description: If C(Op)= 0 then C(GPRn) and C(GPRn+1) are set to zero. No fault results. If C(Op) $\neq$ 0 then C(GPRn) is replaced with the quotent of C(GPRn) divided by C(Op), and C(GPRn+1)is replaced with the remainder.

Indicators affected:
zero - If C(GPRn) = 0 then on; else off
negative - If $C(GPRn_0)$ = 1 then on; else off
carry - If C(GPRn+1) = 0 then on; else off
overflow - If $C(GPRn+1_0)$ = 1 then on; else off

---

Mnemonic: DRLS
Title: Double register left shift
Operation code: 03
Class: read
Group 1 - logical
Description: The operand of the instruction is interpreted to contain two register numbers and a shift count. The following diagram indicates which bits specify these three parameters.



GPRr1 is concatenated to the left of GPRr2 to form a continuous 128 bit register, which is shifted left "S" number of bits. The vacated bits are set to zero. R1 = R2 is legal, allowing a circular shift of a single register.

Indicators affected:
zero - If C(GPRr1) = 0 then on; else off
negative - If a one bit is shifted out of GPRr1$_0$ then on; else off
carry - If C(GPRr2) = 0 then on; else off
overflow - If a one bit is shifted out of GPRr2$_0$ then on; else off

---

Mnemonic: EAPn
Title: Effective address to pointer n
Operation code: 42
Class: none
Group: 2
Description: The effective address of the memory type operand specification is computed. If absolute, an illegal procedure fault is generated. If a (segment number|word number) address, then it replaces C(GPRn).
Indicators affected: none

---

Mnemonic: EORn
Title: EXCLUSIVE OR to register n
Operation code: 04
Class: read
Group: 1 - logical
Description: A bit-wise EXCLUSIVE OR is performed between C(Op) and C(GPRn), with the result replacing C(GPRn).
Indicators affected: see ANDn

---

Mnemonic: LDBR
Title: Load descriptor base register
Operation code: 43
Class: read
Group: 2
Description: Execution allowed in ring 0 only. C(DBR) is replaced by C(Op$_{24-63}$).
Indicators affected: none

---

Mnemonic: LIT
Title: Load interval timer
Operation code: 05
Class: read
Group: 1 - logical
Description: Execution allowed in ring 0 only. C(ITR) is replaced by C(Op$_{32-63}$).
Indicators affected: none

---

Mnemonic: LOADn
Title: Load register n
Operation code: 06
Class: read
Group: 1-logical
Description: C(GPRn) is replaced by C(Op).
Indicators affected: same as ANDn

Mnemonic: MPYn
Title: Multiply register n
Operation code: 23
Class: read
Group: 1 - arithmetic
Description: C(GPRn) is replaced by product of C(GPRn) and C(Op).
Indicators affected: see ADDn

Mnemonic: ORn
Title: OR to register n
Operation code: 07
Class: read
Group: 1 - logical
Description: C(GPRn) is replaced by the bit-wise logical ØR of C(GPRn) and C(Op).
Indicators affected: see ANDn

Mnemonic: RLSn
Title: Register n left shift
Operation code: 10

Class: read
Group: 1 - logical
Description: The last 6 bits of the operand of the instruction are interpreted to contain a shift count. GPRn is shifted left the specified number of bits. Vacated bits are set to zero.

Indicators affected:
- zero - If C(GPRn) = 0 then on; else off
- negative - If C($GPRn_0$) = 1 then on; else off
- carry - If a one bit is shifted out of $GPRn_0$ then on; else off

Mnemonic: RMCn
Title: Restore machine conditions through register n
Operation code: 44
Class: read
Group: 2

Description: The PRR, IR, and GPR0 through GPRn are reloaded from the n+3 words beginning with the operand. PRR is reloaded from the last three bits of Op, IR from the last nine of Op+1, and GPR0 through GPRn from the entire contents of Op+3 through Op+n+2. PRR is only reloaded to a value that is greater than its initial contents. If the reload value is less than or equal to the initial value, PRR is not reloaded. GPR2 is only reloaded if the instruction is executed in ring 0.

Indicators affected: The entire IR is reloaded.

---

Mnemonic: RRSn
Title: Register n right shift
Operation code: 11
Class: read
Group: 1 - logical
Description: The last 6 bits of the operand of the instruction are interpreted to contain a shift count. GPRn is shifted right the specified number of bits. Vacated bits are set to zero.

Indicators affected:
zero - If C(GPRn) = 0 then on; else off
carry - If a one bit is shifted out of $GPRn_{63}$ then on; else off

---

Mnemonic: SDBR
Title: Store descriptor base register
Operation code: 45
Class: write
Group: 2
Description: C(Op) is replaced by C(DBR). Bits 0-23 of the operand are set to zero.
Indicators affected: none

---

Mnemonic: SIC
Title: Set interrupt cell
Operation code: 12
Class: read
Group: 1 - logical
Description: The last four bits of the operand of the instruction are interpreted to contain a port number. The interrupt cell in the memory module connected to the designated port is set on. Execution allowed in ring 0 only.

Indicators affected: none

Mnemonic: SIIS
Title: Set interrupt inhibit switch
Operation code: 13
Class: read
Group: 1 -logical
Description: If the operand contain all zeros then the interrupt inhibit switch is set off. Otherwise, it is set on. Execution allowed only in ring zero.
Indicators affected: none

Mnemonic: SIT
Title: Store interval timer
Operation code: 46
Class: write
Group: 2
Description: C(Op) is replaced by C(ITR). Bits 0-31 of the operand are set to zero.
Indicators affected: none

Mnemonic: STOn
Title: Store register n
Operation code: 47
Class: write
Group: 2
Description: C(Op) is replaced by C(GPRn)
Indicators affected: none

Mnemonic: STOZn
Title: Store register n if operand zero
Operation code: 50
Class: write
Group: 2
Description: The operand is read with the RAH memory command. If it contains all zeros then C(Op) is replaced with C(GPRn). Otherwise C(Op) are not altered.
Indicators affected: zero - If C(Y) initially was 0 then on, else off.

Mnemonic: SUBn
Title: Substract from register n
Operation code: 24
Class: read
Group: 1 - arithmetic
Description: C(GPRn) is replaced by the difference of C(GPRn) and C(Op).
Indicators affected: see ADDn

---

Mnemonic: TIR
Title: Test indicator register
Operation code: 14
Class: read
Group: 1 - logical
Description: C($Op_{55-63}$) are ANDed with C(IR). If the result is zero the next instruction is skipped. The IR is not altered.
Indicators affected: none

---

Date: DEC 16 1968

## Identification

Symbolic instruction representation

## Purpose

For convenience of description, a symbolic representation for instructions is defined. This does not constitute an assembly language for the Clics processor. It is merely a short-hand notation for expressing the contents of an instruction word.

## Representation

An instruction is represented as two parts separated by one or more blanks:

OPNSPEC   OPDSPEC

The OPNSPEC defines the contents of the ØP and R fields, and the ØPSPEC defines the contents of the I,C,SP, and WR fields. The contents of the T field is defined indirectly by the form of OPDSPEC.

An OPNSPEC is just a standard machine instruction mnemonic. If the mnemonic contains the character "n", representing the use of the R field, then this is replaced by a specific integer in the range 0-15. For example, ADD7 would represent the operation "add to register 7". It is also permissable to use the abbreviations ip, tp, cp, sp, and ap in place of 0 through 5 in this way. Thus, EAPO and EAPip are equivalent.

An ØPDSPEC may represent an operand specification of type immediate, register, or memory. The format for each is distinctive, and implies the proper value for the T field of the instruction word.

Immediate operand specifications are enclosed in single quotes. The letter d, b, or o following the quote indicates that the enclosure is to be interpreted as a decimal, binary, or octal number, respectively. Some examples of instructinn representation of this type are:

```
ADDsp     '40'd
LOAD6     '777'0
SUBap     '1101'b
```

All immediate OPDSPECs are padded on the left with zeros to the required 51 bits.

Register operand specifications are signaled by the letters GPR followed by a register number (0 - 15) or one of the five abbreviations described above. Some examples are:

```
LOAD1p    GPRip
ADD6      GPR7
```

Memory operand specifications are more complex. They are also subject to certain informalities. Strictly speaking they should specify the address of the operand in memory in one of four ways, each corresponding to a possible value of the C field.

C = "00"b (signifying that SP and WR specify the segment number and word number) is represented as two numbers separated by a vertical bar, the first specifying SP and the second WR. For example

```
    LOAD6      20|4
```

Frequently names are substituted for the numbers, however, Such names imply no specific mechanical interpretation. They are merely stand-ins for numbers. For example

```
    EAPip    prosegno|saveoffset
```

Presumable what the two names meant would be explained in the text.

C = "01"b (address relative to a pointer register) is specified as a GPR name and numeric offset, separated by a vertical slash. Thus

```
    LOAD10   GPR5|10
    ADD7     GPRap|20
```

Again, sometimes names represent the offset.

C = "10"b (segment number from GPR, word number from instruction) is represented in the same manner, except that a period follows the register name.

```
    LOAD10   GPR5.|10
```

Last and least, C = "11" (absolute) is represented as a natural number.

```
    SDBR     174714
```

If any of these four representations is followed by a ",*" then the I bit is turned "on". Thus,

```
    LOAD15   GPR7.|20,*
```

indicates its operand indirectly.

In some places in this notebook when the actual form of the type memory operand specification is unknown or unimportant to an explanation, just a name for the item to be addressed is substituted. For example

```
    EAPap    arglistptr
```

might be used to signify the placing in ap of the address of the argument list pointer, independent of how that address might be specified.

Date: DEC 16 1968

## Identification

Processor logic organization overview

## Purpose

In this section the general organization of the logic of a processor in the Clics system is described.

## Introduction

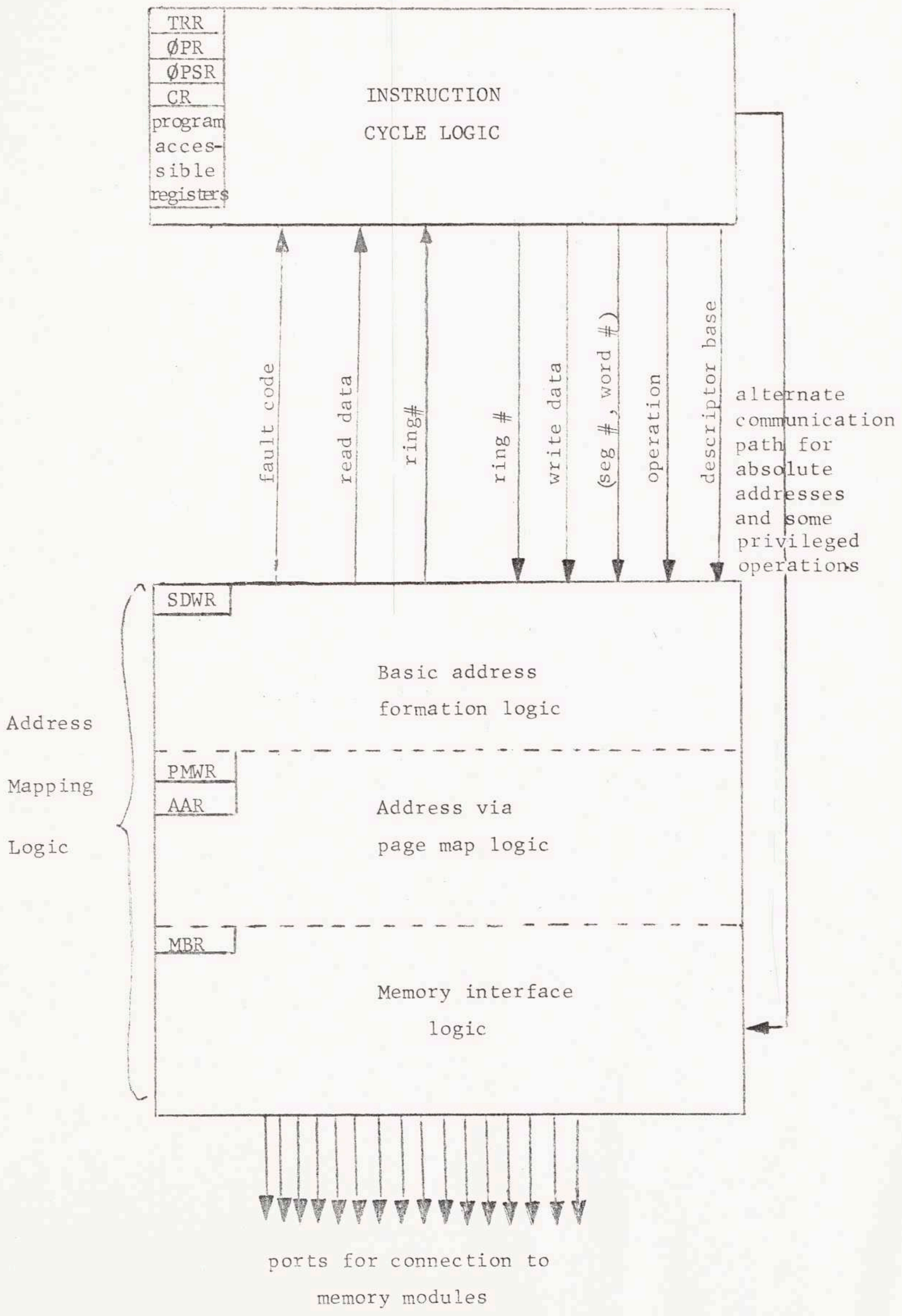
The processor is organized in such a way that the instruction cycle logic (which fetches, decodes, and performs instructions) can view memory as the segmented address space belonging to the process that is currently executing. Three modules of logic, known collectively as the address mapping logic, interposed between the instruction cycle logic and the ports to the memory modules create this effect. The instruction cycle logic normally communicates only with the outermost of these providing (segment number |word number) addresses and operation indicators that specify what operation is to be performed on the words thus identified. The three modules then locate the appropriate segment descriptor word, validate the access, and follow the page maps to the desired word. The operation indicated is performed, and a status code returned to the instruction cycle logic specifying the outcome (successful, or unsuccessful).

## Asynchronous logic

The processors use asynchronous logic. The time needed for each module of logic to perform its task may vary with the specific circumstance. Each task is initiated by the presence of a signal on a control line. Other lines carry input data to the module as level signals. The invoked module communicates the results as level signals on output data lines. Only when a signal is placed on the output control line does the module that initiated the action inspect the results. Each of the lines between modules carries a single bit of information.

## General organization

The diagram on the following page represents the overall organization of the processor logic modules. The large boxes represent the modules, and the small the associated registers. These registers may function as sources or sinks for the level signals on the output or input lines between modules, which are represented as arrows. The pair of control lines for initiating action and signaling termination have been omitted from the diagram.


TRR
ØPR
ØPSR
CR
program accessible registers
INSTRUCTION CYCLE LOGIC
fault code
read data
ring#
ring #
write data
(seg #, word #)
operation
descriptor base
alternate communication path for absolute addresses and some privileged operations
SDWR
Basic address formation logic
Address Mapping Logic
PMWR
AAR
Address via page map logic
MBR
Memory interface logic
ports for connection to memory modules

Processor logic organization

## Identification

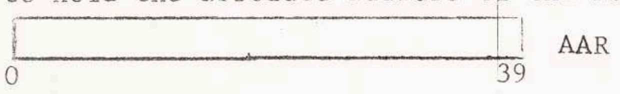
Hidden processor registers

## Purpose

The processors each make use of several hidden registers during the operation cycle. This section describes the format and use of some of these hidden registers. In each case, the indicated field designators match the designators of the entity held.

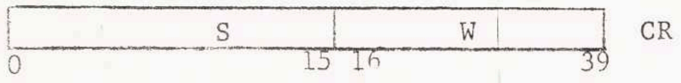
## Absolute address register

The absolute address register is used by the address via page map logic to hold the absolute address of the next memory reference.



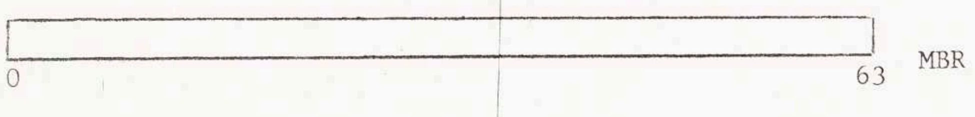
## Call register

The call register is used by the instruction cycle logic to retain the (segment number, word number) address of the location being CALLed.



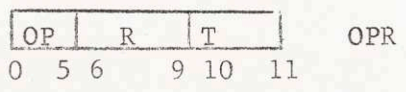
## Memory buffer register

The memory buffer register is used by the memory interface logic to retain each word that is retrieved from memory.



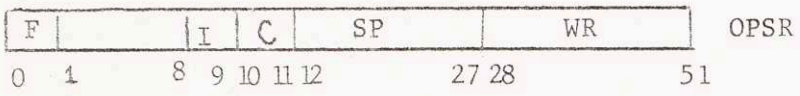
## Operation register

The operation register is used by the instruction cycle logic to hold the operation specification portion of the instruction being executed.



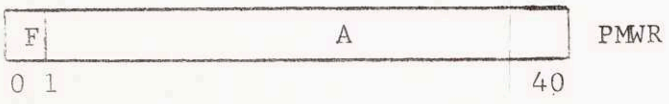
## Operand specification register

The operand specification register is used by the instruction cycle logic to hold the operand specification portion of each instruction word and indirect word referenced.



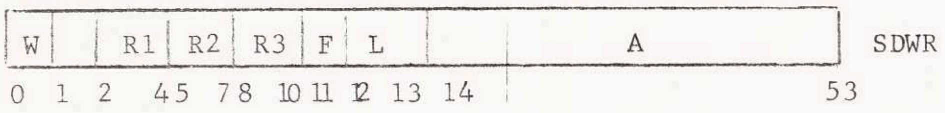
Page map word register

The page map word register is used by the address via page map logic to hold the page map words that it retrieves.



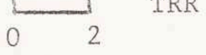
Segment descriptor word register

The segment descriptor word register is used by the basic address formation logic to hold segment descriptor words that it retrieves.



Temporary ring register

The temporary ring register is used by the instruction cycle logic to retain the ring number of each instruction word or indirect word that is referenced.



Date: DEC 16 1968

## Identification

Instruction cycle logic

## Purpose

The instruction cycle logic is the controlling module of the processor. This section describes its organization and functions.

## Introduction

The instruction cycle logic module can be divided into 7 units which operate sequentially. Many of these units invoke the address mapping logic to perform operations on words specified by segment number and word number. They also have access to the final layer of that logic to reference memory by absolute address, and to perform certain privileged group 1 instructions.

## Organization

The diagram on the following page indicates the relationship of the 7 units in the instruction cycle logic. The arrows indicate the flow of control from unit to unit. Each instruction cycle begins in the interrupt checking logic. The fault/interrupt mechanism may be entered from any of the units if a fault condition is noted.

## Interface to address mapping logic

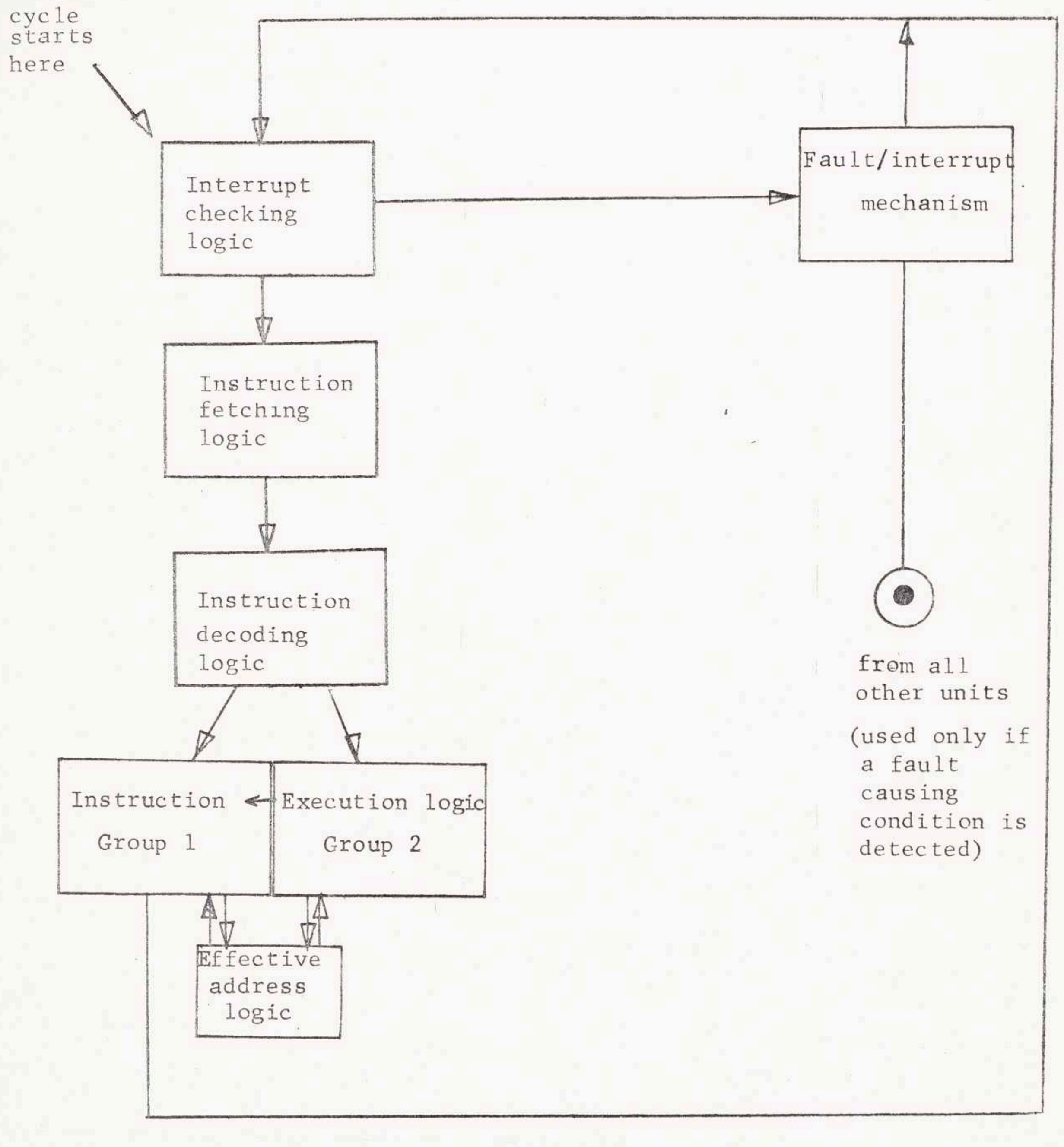
Following is a description of the interface between the instruction cycle logic and the address mapping logic. Several of the units in the former invoke the latter to reference words in the segmented address space of the process executing on a processor.

Five groups of lines carry input data from the instruction cycle logic to the address mapping logic. They are:

descriptor base address: These 40 lines carry the contents of the DBR. This is the absolute address of the page map for the descriptor segment of the process executing on the processor.

operation indicator: These 3 lines indicate the operation to be performed by the address mapping logic. The operations that may be indicated are READ, READ and HOLD, REWRITE, WRITE, and CALL.

segment number and word number: These 40 lines carry the segment number and the word number of the word to be referenced.


cycle starts here
Interrupt checking logic
Instruction fetching logic
Instruction decoding logic
Instruction Group 1
Execution logic Group 2
Effective address logic
Fault/interrupt mechanism
from all other units
(used only if a fault causing condition is detected)

Units of the instruction cycle logic, and flow of control during each instruction cycle write data: These 64 lines carry the word of date to be written if the operation indicator specifies REWRITE or WRITE.

ring number: These 3 lines carry the ring number relative to which permission to perform the indicated operation is to be validated.

Three groups of lines carry output data. They are:

read data: If the operation indicated is READ or READ and HOLD these 64 lines return the word read from memory.

ring number: These 3 lines return the ring number in the R1 field of the access mode indicator of the segment descriptor word of the word READ, or the ring number in the R2 field of the word to be CALLed. A ring number is not returned if the operation indicates is one of the other three.

fault: These 3 lines indicate that the operation was successfully completed, or return the fault code which specifies the reason for the failure.

## Interrupt checking logic unit

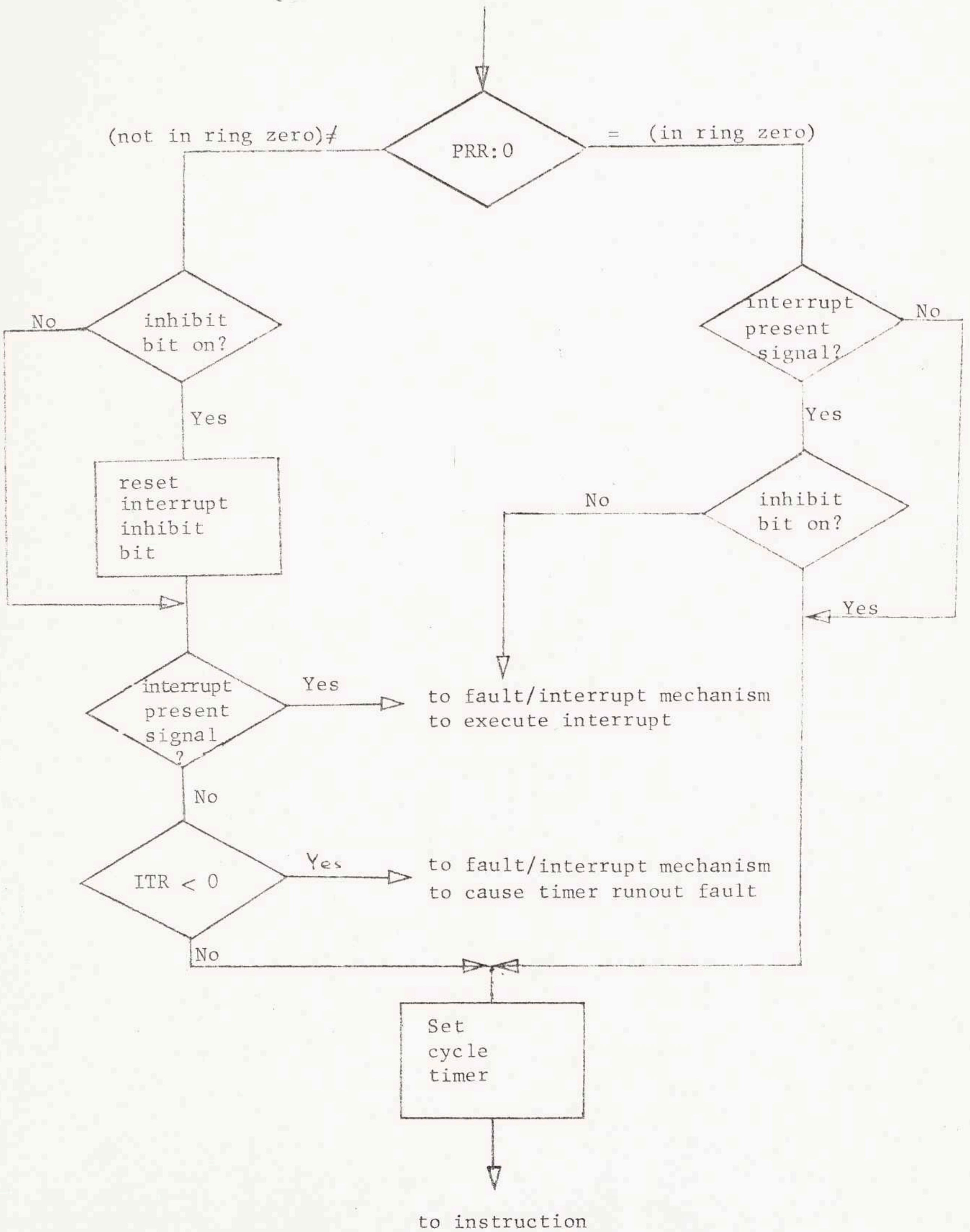
The interrupt checking logic unit begins each instruction cycle, and is responsible for detecting interrupts and timer runouts. The following flow diagram indicates its basic logic. The procedure ring register is checked to determine if the processor is running in ring zero, or not. If so, the interrupt present lines of the major module interfaces are checked for a signal indicating that the interrupt cell in the attached memory module is set. If a signal is found, and interrupts are not inhibited, control is passed to the fault/interrupt mechanism to execute the interrupt. If the processor is running outside of ring zero, the inhibit switch is checked and set "off" if "on", and then the interrupt present lines are checked. Any signal causes control to pass to the fault/interrupt mechanism to execute the interrupt. If no signal is found, the interval timer is inspected for a negative value which causes a timer runout fault. The final step when operating in any ring, assuming no faults or interrupts, is to set the instruction cycle timer with a positive value and pass control to the instruction fetching logic. This timer is decremented each time the ITR is decremented. If the cycle timer becomes zero a lockup fault is asynchronously generated, preventing a loop for indirect words from locking the processor.

## Fault/interrupt mechanism

This unit is described in the separate section that specifies the fault/interrupt mechanism for the processors. It obtains control only if an interrupt present signal is detected or a fault condition occurs.

## Instruction fetching logic unit

Assuming successful transition of the interrupt checking unit, the


PRR:0
(not in ring zero)≠
= (in ring zero)
inhibit bit on?
No
Yes
reset interrupt inhibit bit
interrupt present signal?
Yes
to fault/interrupt mechanism to execute interrupt
No
ITR < 0
Yes
to fault/interrupt mechanism to cause timer runout fault
No
interrupt present signal?
No
Yes
inhibit bit on?
No
Yes
Set cycle timer
to instruction fetching logic

Interrupt Checking Logic Flow Diagram

instruction fetching logic unit is the next to gain control. It is this unit's responsibility to retrieve from memory the next instruction word to be performed. The segment number and word number address of the instruction is taken from ip (GPRO). The ring number provided for validation purposes is taken from the procedure ring register. The address mapping logic is invoked to retrieve the instruction, using the the operation indicator READ. A returned fault code is passed to the fault/interrupt mechanism. Successful retrieval is followed by placing the operation specification portion of the instruction word in ØPR and the operand specification portion in ØPSR. The returned ring number is ignored.

<u>Instruction decoding logic unit</u>

Assuming successful transition of the instruction fetching unit, the instruction decoding logic is the next to gain control. This unit separates instructions from group 1, and 2; and determines that the type of the operand specification is valid for the group. It also checks the fault bit of the instruction word. If an invalid type specification is discovered, or the fault bit is on, the fault/interrupt mechanism is given control with the proper fault code provided as input. Otherwise, control is passed to the instruction execution logic unit for the proper group.

<u>Group 1 instruction execution logic unit</u>

Instructions in group 1 have the common characteristic that they read their operand. The initial step of the group 1 instruction execution logic is to load the temporary ring register (TRR) from the procedure ring register. Next, the validity of the register specification in the "R" field of the instruction is determined. (If invalid an "illegal procedure" fault is caused.) The "T" field is then inspected to determine the type of the operand specification. If <u>register</u> or <u>immediate</u> then the operand is readily available. If <u>memory</u>, several steps must be taken to retrieve it. They are outlined below:

a. The effective address logic unit is given control, accepting as input the operand specification in ØPSR and the ring number in TRR. Upon return (if no faults occurred) GPR1 (tp) will contain the segment number and word number of the operand, and TRR the ring from which the last indirect word referenced was obtained. (In special circumstances GPR1 will contain an absolute address. Processing of this case, while not complex, is not considered here.)

b. The address mapping logic is invoked to obtain the operand word from memory. Provided as input are the (segment number |word number) address from GPR1, the DBR image, the ring number in TRR, and the command READ. Upon return the <u>read data</u> lines will contain the operand word.

When the operand of any of the three types becomes available the instruction specified by the operation code in ØPRop is performed. The ip (GPRO) is properly reset to indicate the next instruction to be executed and control is returned to the start of the instruction cycle. If any fault conditions are detected control is immediately passed to the fault/interrupt mechanism with the proper code.

The execution of the SIC instruction requires a secondary reference to a memory module. In this case, the last four bits of the operand is interpreted to contain a port number, identifying a port on the processor executing the instruction. The memory module connected to this port is sent any SIC command by the memory interface portion of the address mapping logic.

## Group 2 instruction execution logic unit

Group 2 instructions do not make a common type of operand reference, thus the individual instructions must be separated before the reference can be made. Validation of specification, and determination of the effective address of the operand can be accomplished the same way for all, however, since all require operand specifications of type memory. The steps involved are the same as those for group 1 instruction with memory type operand specifications.

Once the effective address is available in tp (GPR1) and the ring number is available in TRR the specified instruction is performed. Performance of each makes various demands on the address mapping logic. The following table specifies the operation indicators input for each group 2 instruction. The segment number and word number, and ring number from tp and TRR are also input.

| instruction | operation indicator |
|---|---|
| ADDSn | READ and HOLD followed by REWRITE |
| CALLn | CALL, READ, and repeated WRITEs |
| EAPn | the operand is not referenced |
| LDBR | READ |
| RMCn | repeated READs |
| SDBR | WRITE |
| SIT | WRITE |
| STOZn | READ and HOLD followed by REWRITE |

As can be seen from the table, CALLn is a very special instruction. While functionally described in the instruction set description, certain steps in its performance require further operational specification. Once the effective address and "TRR" ring number become available from the effective address logic, the address mapping logic is invoked with the operation indicator CALL. A successful return produces the "R2" ring number of the location to be called, if it can be called from the ring whose number was input from TRR.

(Incidentially, TRR may specify a higher ring than that specified by the current PRR if indirection was encountered in forming the effective address.) If this returned number is higher than PRR, then PRR is saved in the TRR as the ring number of the ring to be called. if it is lower than PRR, then the returned number is saved. This prevents an outward call from occurring if the processor is currently executing underneath the "R2" execute bracket of the procedure called. Eventually, the ring number saved in TRR will replace PRR.

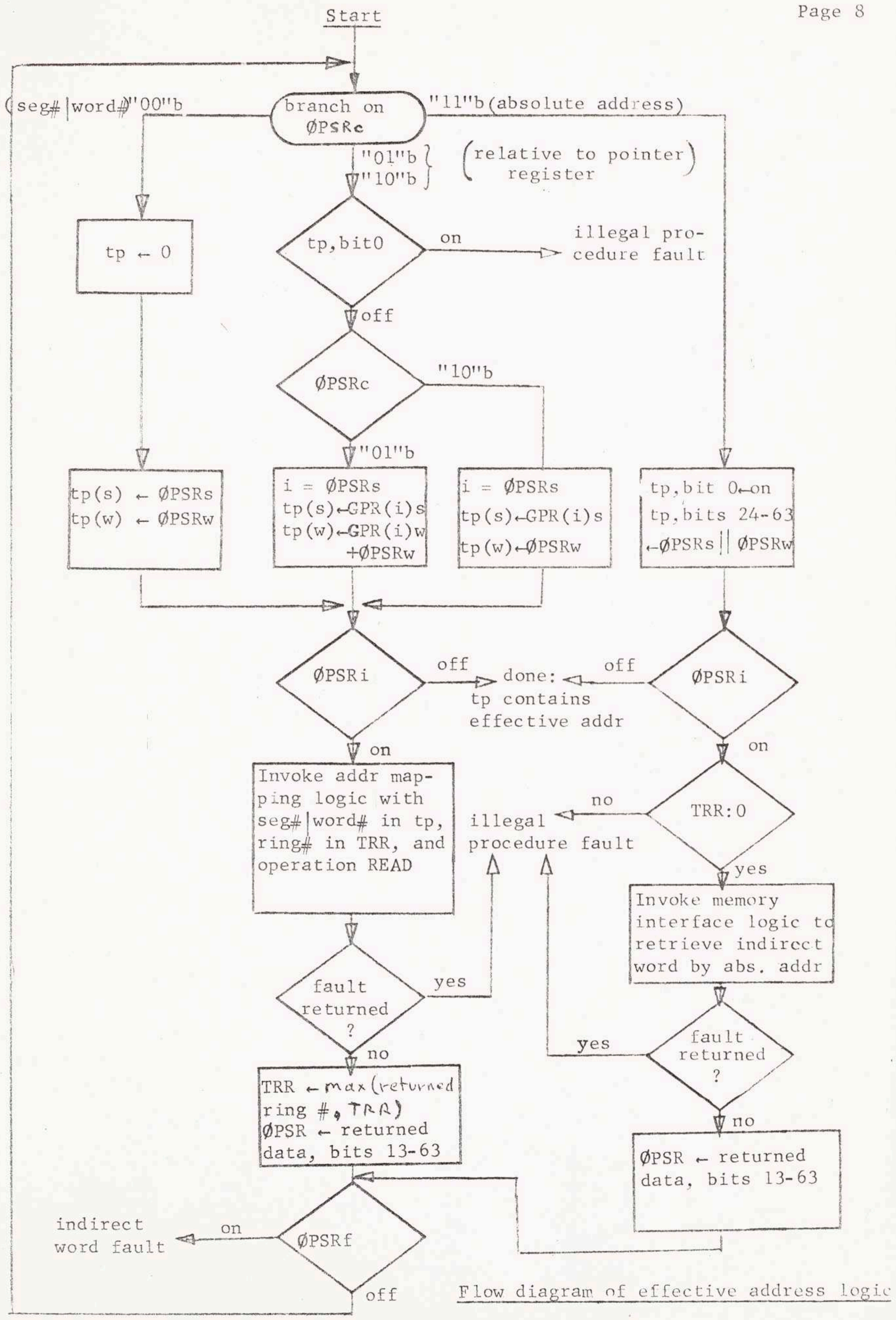
## Effective address logic unit

The effective address logic unit is used by the execution logic for group 1 and 2 instructions to compute the effective address of the operand if it is a memory word. The effective address, normally a segment number and a word number, is returned in tp (GPR1). If it happens to be absolute, then it is returned in tp, and bit 0 is set to 1.

This unit obtains its input arguments from processor registers. ØPSR is expected to contain the operand specification of type memory that is to be evaluated. TRR is expected to contain the ring number to which access validation is to be relative. tp is expected to contain the segment number and the word number of the instruction word that contained the operand specification.

The address mapping logic is used to retrieve any indirect words that may be involved in the evaluation. If an indirect word must be retrieved by absolute address, the memory interface portion is directly invoked. At completion, TRR contains R1 the ring number of the last indirect word referenced. This is the ring number passed in to the address mapping logic for access validation purposes.

The flow diagram on the following page indicates the operation of this unit. Abbreviations used are specified in a separate notebook section



Flow diagram of effective address logic

## Identification

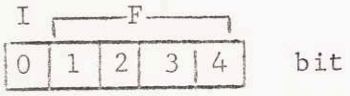
Fault/interrupt mechanism of instruction cycle logic

## Purpose

The fault/interrupt mechanism in the instruction cycle logic receives control for two reasons: an interrupt present line signal has been noticed by the interrupt checking logic when the interrupt inhibit switch is "off", or a fault condition has been detected by some module in the instruction cycle logic. The purpose of this mechanism is to process the fault or interrupt by making a hardware call to the fault/interrupt interceptor code whose location is specified by GPR2.

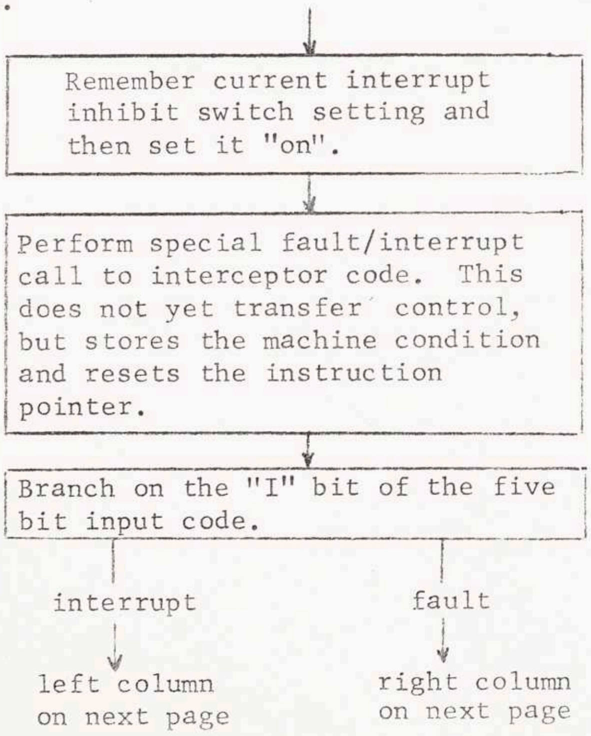
## Operation

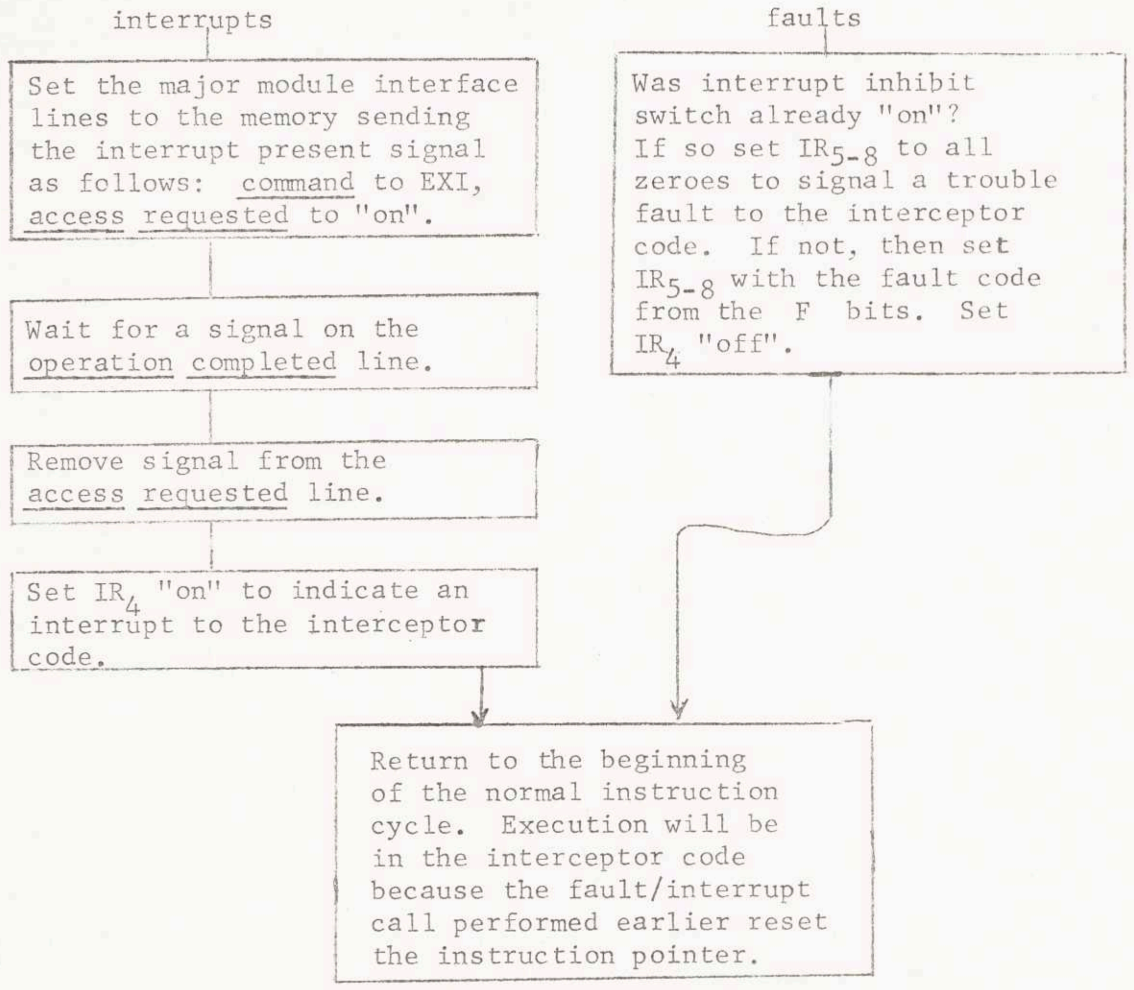
When this mechanism is invoked a five bit code is presented. The format is indicated below:



The "I" bit indicates whether an interrupt signal or a fault has occurred. If "off" a fault is the case, and the F bits contain a code identifying the fault. (The codes are specified later in this section.) If "on" an interrupt signal is the case and the F bits can be ignored.

The following diagram specifies the action taken by the fault/interrupt mechanism when invoked.


Remember current interrupt inhibit switch setting and then set it "on".
Perform special fault/interrupt call to interceptor code. This does not yet transfer control, but stores the machine condition and resets the instruction pointer.
Branch on the "I" bit of the five bit input code.
interrupt
fault
left column on next page
right column on next page


interrupts

Set the major module interface lines to the memory sending the interrupt present signal as follows: command to EXI, access requested to "on".

Wait for a signal on the operation completed line.

Remove signal from the access requested line.

Set $IR_4$ "on" to indicate an interrupt to the interceptor code.

faults

Was interrupt inhibit switch already "on"? If so set $IR_{5-8}$ to all zeroes to signal a trouble fault to the interceptor code. If not, then set $IR_{5-8}$ with the fault code from the F bits. Set $IR_4$ "off".

Return to the beginning of the normal instruction cycle. Execution will be in the interceptor code because the fault/interrupt call performed earlier reset the instruction pointer.

## Fault/interrupt call

The fault/interrupt call performed as the second step of processing a fault or an interrupt signal is very similar to the action taken by the processor when it executes a CALL15 instruction. There are two differences: the instruction pointer (GPR0) is not incremented by one before being saved, and the stack frame used is located in the ring zero procedure stack 18 locations beyond the word pointed to by word zero of that stack. (The normal call uses a stack frame that is located at the word pointed to by the zeroth word of the stack.) The former is necessary so that when the inteceptor code returns the faulting or interrupted instruction can be re-executed, and the latter so that if the fault or interrupt has occured between the execution of a CALL15 instruction into or in ring zero and the associated save sequence, the new frame will not be destroyed. The location called by the fault/interrupt call is specified by segment and word number in GPR2. Since the effective address is immediately available the effective address logic need not be invoked. This also prevents destroying the current contents of tp (GPR1).

## Faults

Following is a list of the 10 possible faults, their causes, and the associated fault codes.

trouble: 0000 A fault has occured while the interrupt inhibit switch is on.

lockup: 0001 Execution of a single instruction has taken more than a predetermined number of memory refernces. The specific number is alterable by switch setting on each processor.

illegal memory command: 0010 A returned status of "illegal action" or "unknown address" has been received from a referenced memory module, or reference was made through a port not connected to any memory module.

parity: 0011 A returned status of "parity error " has been received from a referenced memory module.

instruction word: 0100 The fault bit in an instruction word was set on.

indirect word: 0101 The fault bit in an indirect word was set on.

segment descriptor: 0110 The fault bit in a segment descriptor word was set on.

page map: 0111 The fault bit in a referenced page map word was set "on".

illegal procedure: 1000 An illegal specification was encountered in an instruction or an indirect word, or an access mode indicator violation was attempted.

timer runout: 1001 The interval timer register has become negative and the processor is not operating in ring 0.

## Identification

Address mapping logic

## Purpose

The three logic units that are collectively known as the address mapping logic jointly perform three primary functions: the validation of requested memory accesses, the transformation of (segment number|word number) addresses into absolute addresses, and the performance of the indicated memory reference. This section describes the organization of these three units and their operation.

## Introduction

In the section providing the overview of the processor logic organization we mentioned that the three units in the address mapping logic had the cumulative affect of making memory appear to the processor as the segmented address space of the executing process. This is the view at the interface to the outermost layer, the basic address formation logic. The details of this interface were provided in the previous section.

The view at the interface to the next layer, the address via page map logic, is more primitive. Words are addressed by giving the absolute base address of a segment, the number of levels of page maps for the segment, and the offset within the segment. At the interface to the final layer, the memory interface logic, the view is of a contiguously addressable memory, and addressing is by absolute location. The memory interface logic is the only unit to accept memory in its raw form, a collection of up to 16 separate memory modules each containing a group of addresses.

The diagram on the next page illustrates the layers of address mapping logic and the interfaces between them.

## Internal interfaces

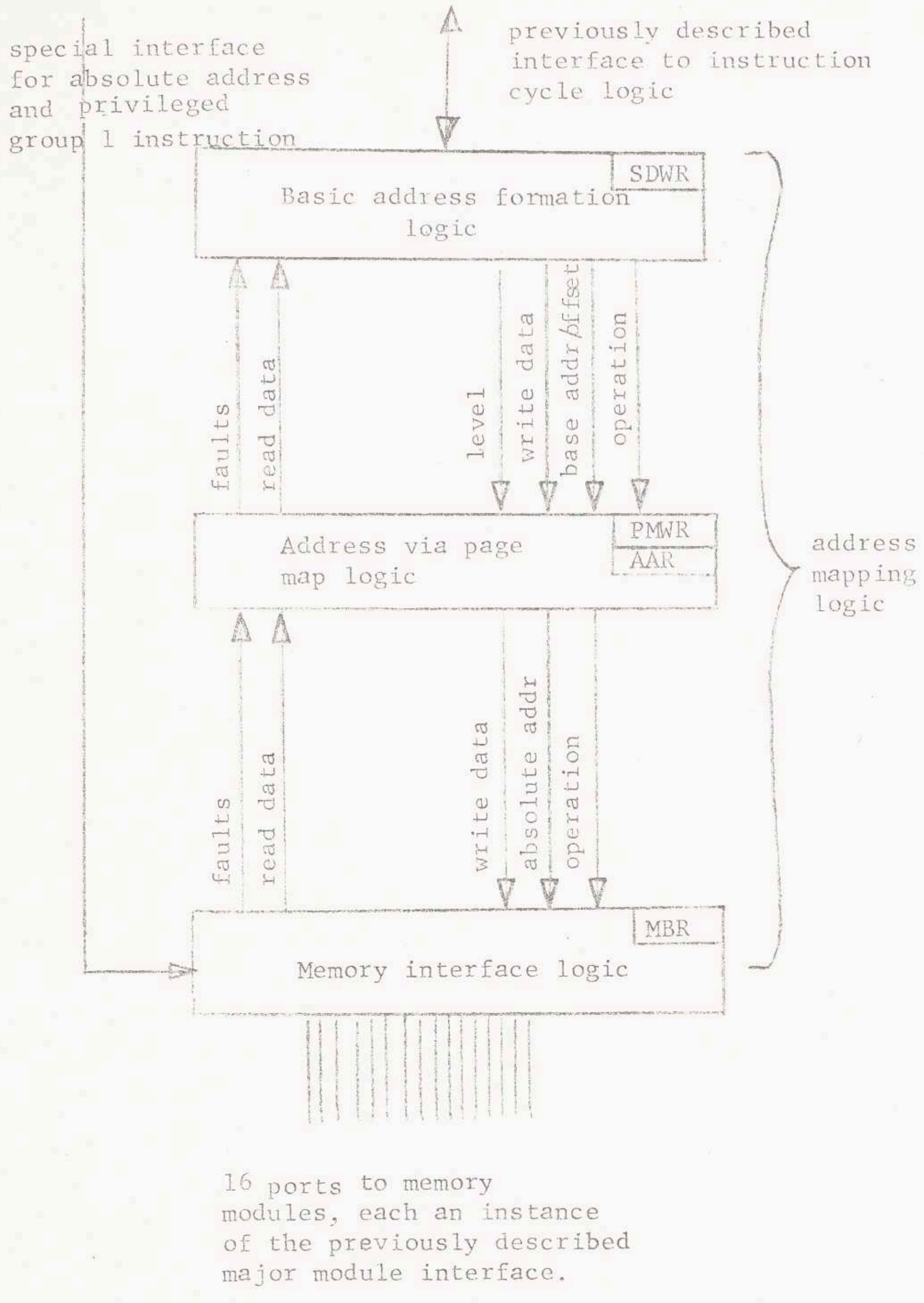
Following is a description of the interfaces between the basic address formation logic and the address via page map logic, and the address via page map logic and the memory interface logic.

The input lines to the address via page map logic are:

operation indicator: These 2 lines specify whether the location specified is to be written, read, or referenced with either half of a read/alter/rewrite cycle. The commands are READ, WRITE, READ and HOLD, and REWRITE.

segment address and offset: These lines carry the base address of a segment (the absolute address of the highest level page map) and the offset within the segment for the word to be referenced.)

level: These 2 lines carry the number of levels of page maps for the segment specified above.



Units of the address mapping logic, and the internal interfaces write data: If the operation indicator specifies WRITE or REWRITE these 64 lines carry the word of data to be written into memory.

The output lines from the address via page map logic are:

read data: If the operation indicator specified READ or READ and HOLD then these 64 lines carry the data word read from memory.

fault code: These lines indicate successful completion of the requested operation, or carry a code specifying the reason for failure.

The input lines to the memory interface logic are:

operation indicator: same as operation indicator lines described above, except that REWRITE is missing (WRITE performs this function).

absolute address: These 40 lines carry the absolute address of the memory word to be referenced.

write data: same as write data lines described above.

The output lines from the memory interface logic are the same as the output lines from the address via page map logic.

## Basic address formation logic

As described in the previous section, five operation indicators may be input to the basic address formation logic. These are listed and described below.

1. READ - The word specified by segment number and word number is to be read from memory. Permission to read the word is to be validated relative to the ring number provided as input, and the highest ring number in which the word may be written is to be returned (specified by R1 in the segment descriptor word of the word to be read), along with the contents of the word itself.
2. READ and HOLD, and REWRITE - These two operations are always specified one after the other. The first causes the word whose segment number and word number is given as input to be read from memory, subject to permission to write in the word from the ring number also given as input. The second causes the input data word to be written back into the same location. No access validation is performed with the second cycle. The RAH and WRITE memory commands are eventually used, locking the word against access from another processor between operations.

3. WRITE- The word given as input data is to be written into memory at the location specified bysegment number and word number, subject to permission to write in the word from the ring whose number is also given as input.

4. CALL - The segment number and word number given as input is that of the operand of a CALLn instruction. Only the SDW for the containing segment need be retrieved. Permission to CALL the specified location is verified, and the "R2" ring number of the CALLed location is returned.

The operation of the basic address formation logic is best understood by cases. The simplest (but most obscure) case is the response to the input operation indicator REWRITE. Since no access validation is associated with this operation and since the absolute address of the location to be written is still available in the ARR (see READ and HOLD below) of the address via page map logic, the write data provided as input is passed on to the write data input of the address via page map logic, and this unit is invoked with the command REWRITE. Any returned fault codes are then passed back to the caller (instruction cycle logic).

The response to the operation indicator CALL requires that the SDW of the segment specified be retrieved. This is done by invoking the address via page map logic, providing as input the operation indicator READ, the segment address and offset that is the DBR image and segment number received as input, the level number of 1, and no write data. If no faults are returned, the retrieved SDW is placed in SDWR. The CALL access validation is then performed and the resulting ring number returned to the instruction cycle logic. If the access requested is not allowed, a fault code is returned instead. The access check for the CALL operation, and also for the operations discussed below, is described in the notebook section on the access checking logic.

The remaining operation indicators are processed alike, except for the specific access check involved and the command passed to the address via page map logic. The required SDW is first retrieved, as described above, and the proper access check made. The address via page map logic is then invoked a second time to make the desired reference to the specified location. The base address for the segment is taken from the "A" field of the retrieved SDW, and the level number is taken from the "L" field. The word number provided as input is the necessary offset. The input operation indicators READ, READ and HOLD, and WRITE, are translated into the same operation indicators for the address via page map logic. Again, returned faults are passed on to the instruction cycle logic.

## Address via page map logic

The address via page map logic may be invoked with four different operation indicators: READ, WRITE, READ and HOLD, and REWRITE. These specify the type of reference to be made to the word specified by the other input lines This unit is responsible for locating a word within a segment, once the absolute address of the highest level page map for the segment is known (and the number of levels of page maps). It operates on the principle that the first, middle, and last eight bits of the 24 bit word number are the master page map word number, the page map word number, and the word number within the page of the word to be referenced, respectively. Depending on the level number of the segment; one, two, or three requests are made to the memory interface logic to retrieve the necessary page map words and the word referenced. Retrieved page map words are retained in PMWR so that their fault bits may be checked, and their address fields referenced. The absolute address of the next memory location to be referenced is formed in AAR.

The operation indicators READ, WRITE, and READ and HOLD are translated to the same operation indicators for the memory interface logic when the reference to the data word is made. The operation indicator REWRITE is handled somewhat differently than the others. Since the REWRITE always immediately follows a READ and HOLD, the ARR already contains the absolute address of the location to be referenced. Thus, no page map words need be retrieved. A WRITE operation indicator is passed to the memory interface logic, along with the absolute address in AAR.

As was the case with the basic address formation logic, generated or received faults are passed back to the caller as non-zero fault codes.

## Memory interface logic

The memory interface logic performs the actual communication with the memory modules. The input operation indicators are READ, READ and HOLD, and WRITE. If the operation is READ or READ and HOLD the absolute address provided as input is inspected to select the port of the processor which connects to the memory module containing the address. The memory command READ or RAH is then sent over the lines to that module. When the command is performed, the read word is returned on the lines from the memory module, and the memory interface logic places the 64 bits of information in the MBR. This register is then connected to the read data output lines to provide the signals that communicate the data to the address via page map logic, and further. If an error occurs, a fault code is returned. If the operation indicator specifies WRITE the write data provided as input is sent to the appropriate memory module and the memory command WRITE indicated. All communication to and from the memory modules is via the major module interface

The order in which signals are placed on the lines of the major module interface connected to the selected part, and their duration is described below.

1. Signals representing the address and command are set. If the operation involves active module to passive module data transfer, the data signals are also set.

2. A signal is set on the access requested line.

3. When a signal appears on the operation completed line the code on the status line is noted.

4. If the operation involved passive module to active module data transfer, the data line signals are used to set the MBR.

5. The signal on the access requested line is removed.

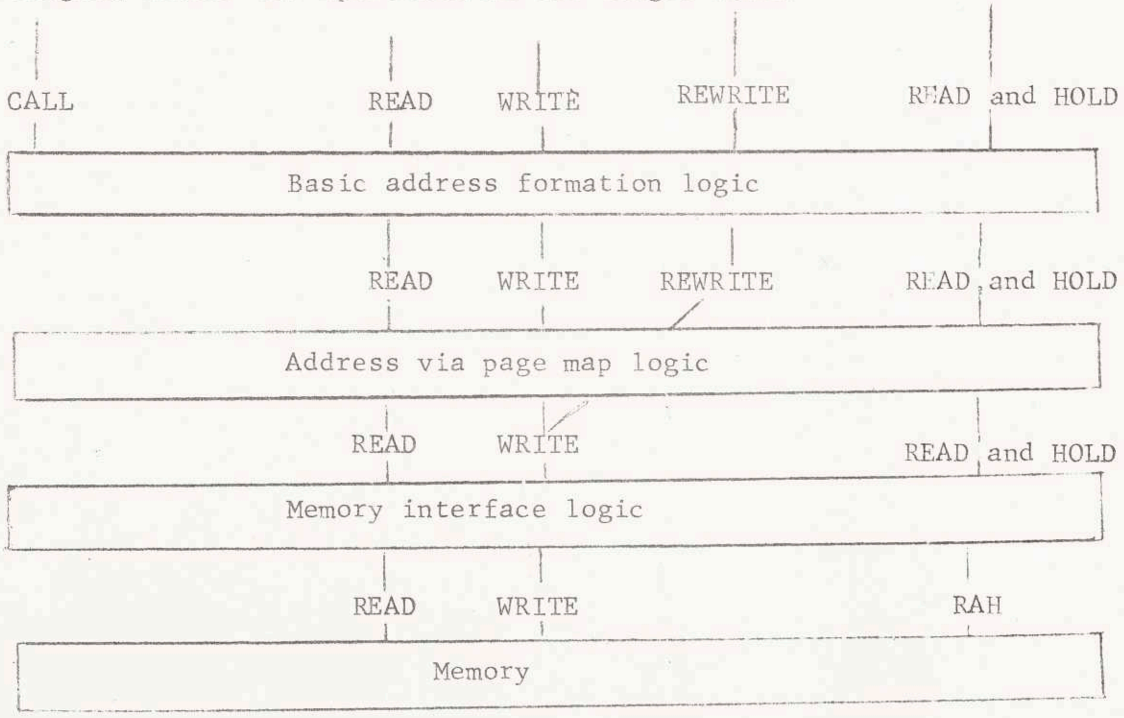
6. Signals set in step 1 are removed.

The removal of the access requested signal in step 5 is the indication for the passive module that the output has been received.

The instruction cycle logic also has a direct connection to the memory interface logic. This is used to retrieve operands and indirect words by absolute address, and to perform the SIC instruction which references a register in the memory modules. The interrupt mechanism also uses this connection to sample the interrupt present lines.

## Summary of commands

The following diagram indicates the transformation of commands through the layers of the address mapping logic to memory. In each case the relevant SDW must be READ, and this is not indicated. Rather, the diagram traces the operation on the target word.



## Identification

Access checking logic

## Purpose

The access checking logic is part of the basic address formation logic. This section describes its operation.

## Introduction

The access validation described in this section occurs during the operation of the basic address formation logic immediately after the SDW of the segment to be referenced has been placed in the SDWR. The input arguments that are available include an operation indicator that specifies the type of access validation to be performed, and the ring number to which the validation is to be made relative.

The SDW for each segment in a process' address space contains an access mode indicator. This indicator specifies the type of access that is permitted to the corresponding segment (read, and read/write) and the rings from which that access is permitted. Since the SDW of a segment must be referenced by the basic address formation logic each time that a word in the segment is accessed, this module may easily validate the access at each request.

## Access mode indicator

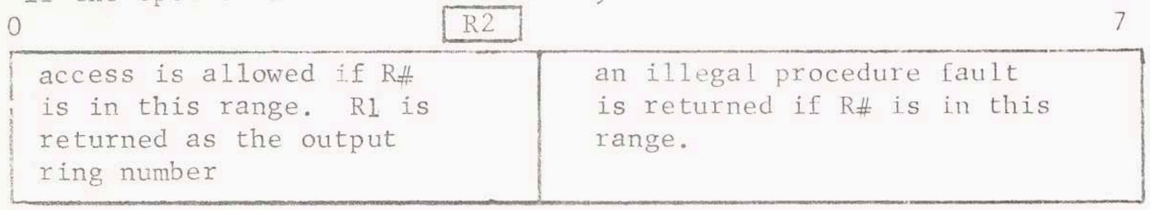
The access mode indicator is the first 11 bits of SDW. The write permit bit specifies the type of access permitted to a segment according to the following table:

| Write permit bit | Access type |
|---|---|
| 0 | read only |
| 1 | read/write |

Note that read permission is implied by the presence of a SDW for a segment in a process' descriptor segment,and that read permission is all that is necessary to execute in a segment.

The three ring indicators in the access mode indicator specify from what rings the specific types of access are allowed. The ring number input from the instruction cycle logic is compared with a subset of these ring indicators (the specific comparisons dependent upon the access requested) and access is permitted if the comparisons meet the criterion indicated by the following tables. The boxes represent the rings (from 0 to 7) in which a procedure or a data segment may reside. The vertical lines indicate the contents of the relevant ring indicators from the access mode indicator in the SDW that is in the SDWR. The symbol "R#" represents the ring number provided as input by the instruction cycle logic.

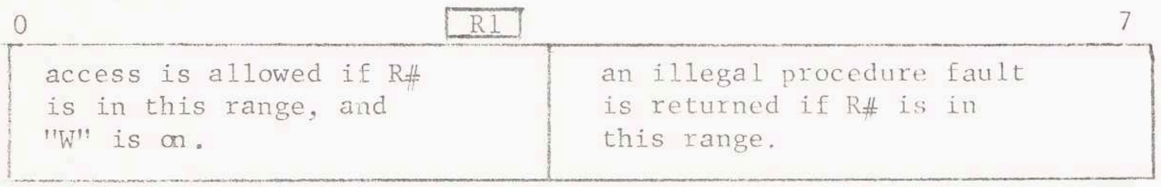
If the operation indicator is READ , then



access is allowed if R# is in this range. R1 is returned as the output ring number

an illegal procedure fault is returned if R# is in this range.

If the operation indicator is WRITE or READ and HOLD, then

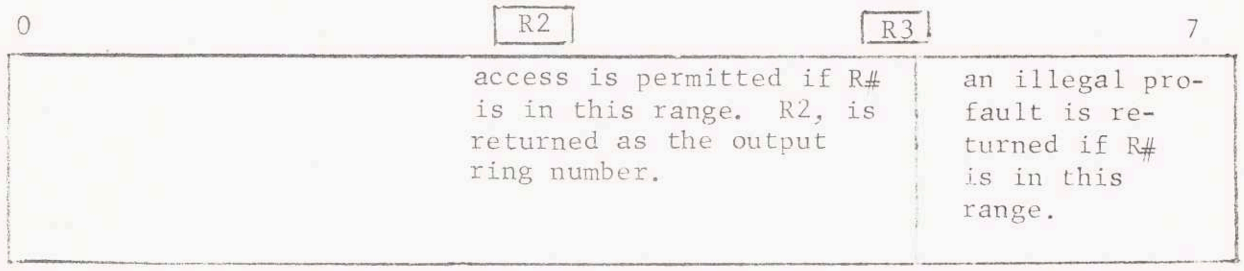


access is allowed if R# is in this range, and "W" is on.

an illegal procedure fault is returned if R# is in this range.

If the operation indicator is REWRITE, then no access validation is necessary.

If the operation indicator is CALL, then



access is permitted if R# is in this range. R2, is returned as the output ring number.

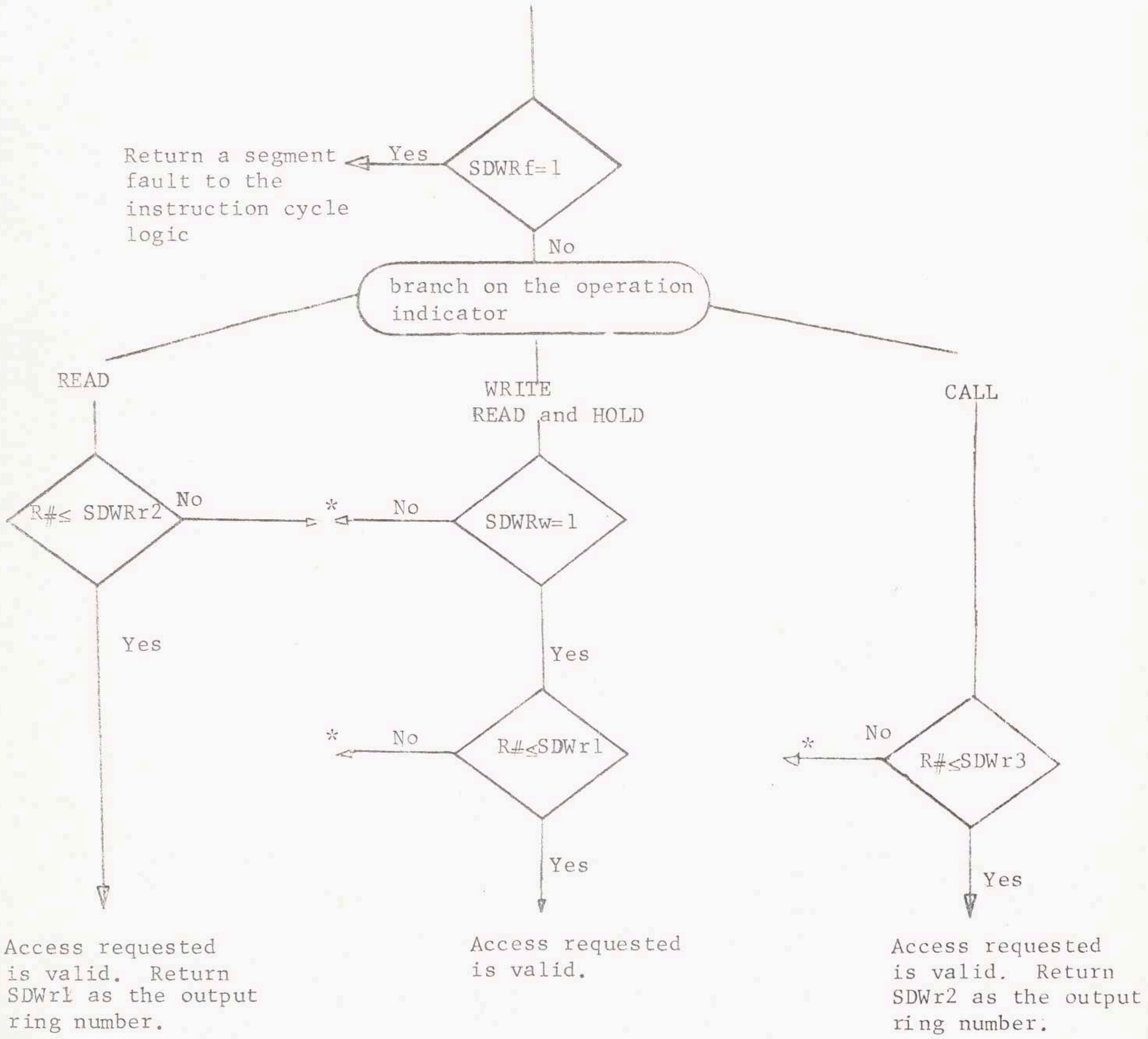
an illegal pro-fault is re-turned if R# is in this range.

Note = If the "W" bit is not on when it is specified that it must be, an illegal procedure fault is returned. This occurs even if R# is the correct range.

Thus, R1 specifies the highest ring from which the segment maybe written (the write bracket), R2 the highest from which it may be read or executed (read and execute bracket) and R3 the highest from which it may be called (call bracket).

## Operation

The following flow diagram describes the steps taken by the access protection portion of the basic address formation logic. Recall that SDWR contains the SDW for the segment in question, and that R# is the ring number input from the instruction cycle logic.


Return a segment fault to the instruction cycle logic
Yes
SDWRf=1
No
branch on the operation indicator
READ
WRITE
READ and HOLD
CALL
R#≤ SDWRr2
No
*
No
SDWRw=1
Yes
Yes
*
No
R#≤SDWr1
*
No
R#≤SDWr3
Yes
Yes
Access requested is valid. Return SDWr1 as the output ring number.
Access requested is valid.
Access requested is valid. Return SDWr2 as the output ring number.

* indicates that an illegal procedure fault is returned to the instruction cycle logic.

Date: DEC 16 1968

## Identification

Directory of processor abbreviations

## Purpose

The description of the Clics processor uses abbreviation to indicate many things. In particular, register names are abbreviated, as are the fields of the register. The scheme used is to designate register with a string of capital letters which are the first letters in theregister's name. The string is then followed by the field designator, in small letters. Subscripts indicate bits within a register. Bit positions are numbered left to right beginning with zero. For example, "GPR1s" would mean general purpose register 1, the segment number field; and "$GPR1_{0-5}$"would mean the left most 6 bits of that register.

## Tables

Following is a table of the register abbreviations used:

| abbreviation | register name |
|---|---|
| AAR | Absolute address register |
| CR | Call register |
| DBR | Descriptor base register |
| GPR0 - GPR15 | General purpose registers 0-15 |
| IR | Indicator register |
| ITR | Interval timer register |
| MBR | Memory buffer register |
| ØPR | Operation register |
| ØPSR | Operand specification register |
| PMWR | Page map word register |
| PRR | Procedure ring register |
| SDWR | Segment descriptor word register |
| TRR | Temporary ring register |

In addition, there are commonly used names for GRR0-GRR5. The following table indicates these:

| GPR | Abbreviation | Name |
|---|---|---|
| 0 | ip | instruction pointer |
| 1 | tp | temporary pointer |
| 2 | cp | call pointer |
| 3 | sp | stack pointer |
| 4 | lp | linkage pointer |
| 5 | ap | argument pointer |

The instruction mnemonics are specified elsewhere, as are the mnemonics for memory commands, and the operation indicators for the modules of the address mapping logic.

## Identification

Input/output controller overview

## Purpose

The input/output controllers (IOC) are special purpose active modules that are specifically designed to control the transfer of information in both directions between memory and the I/O devices. This section provides an overview of the organization and use of the IOC's.

## Introduction

From the standpoint of system interconnection the IOC's are treated the same way as the processors. Each unit of both types is connected via its ports and the major module interface to all the memory modules in the hardware system. In contrast to the processors, however, the IOC's are attached to a second set of units - the I/O devices. Each magnetic recording tape handler and typewriter terminal is connected to an IOC; either directly or indirectly via a switching network. It is the purpose of the IOC's to control the transfer of data from these devices to memory, and from memory to these devices.

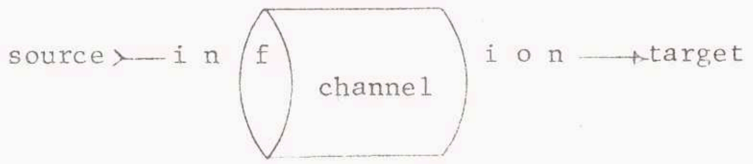
One significant aspect of this data movement is the manner in which source or target memory locations are specified. In keeping with the method used elsewhere in the system, they are given as segment numbers and word numbers. In fact, all references by an IOC, whether for data movement or control purposes, are made by segment number and word number. Of course, for the segment numbers to be meaningful they must be considered in the context of some process' address space, for each is simply an offset in the proper descriptor segment. For this purpose each IOC contains a descriptor base register which contains the absolute location of the base of a descriptor segment. All segment numbers used by an IOC are relative to this descriptor segment which defines an IOC process that consists entirely of data segments. These segments are the sources and targets of data transfers, and the scratch areas for the control functions of the IOC. They may also be included in the address space of other processes wishing to perform I/O.

In order to perform the address mapping associated with addresses containing segment numbers the IOC's also contain address mapping logic just like that of the processors. This logic can be presented with the contents of the IOC descriptor base register, a (segment number | word number) address, a ring number of zero, and an operation indicator; and will compute the absolute location and make the requested reference - just as it did for the processors.

The IOC's also share with the processors the ability to accept and respond to interrupt signals sent through the memories. An IOC may be the control module for a memory, and, as such, receive the interrupt present signal when the interrupt cell in that memory is set. The response is to retrieve a single word of control information from a predetermined memory location. They may also send interrupts (that is, cause the interrupt cell in a memory module to be set), although their capacity to do so is restricted to signalling a specific memory which is specified by switch setting in the IOC.

## Organization

The basic unit of organization in an IOC is the channel. A channel is a device for controlling the transfer of information from a source to a target.



Obviously, in the case of an IOC the most basic type of channel will be that which controls the transmission of information from memory to an I/O device, and in the opposite direction. This is called a <u>data channel</u> and there is one directly attached to each interface to a typewriter terminal or a tape handler. The information transmitted includes both the data to be output or being input, and commands to operate the attached I/O device. Each data channel can be programmed in a very limited way, for each time one is activated it retrieves and executes a single instruction from a prespecified memory location. The instruction, called a data channel word (DCW), contains a command, a segment and word number address, and a count. The command specifies the action, the address the source or target memory location, and the count the number of items to be transfered.

A second type of channel, the <u>list</u> channel, becomes necessary to use a data channel with facility. A list channel controls the transfer of instructions to each data channel. Each data channel is paired with its own list channel. The list channel transfers DCW's to the instruction pickup location for the data channel, and activates the data channel for each. The data channel informs the list channel each time an instruction has been completed, at which point the latter has the option of reactivating the former with a new DCW. The list channels may also be programmed. Each time one is activated it retrieves a single instruction, called a list channel word (LCW), from a prespecified memory location. The LCW specifies a command, a segment and word number address, and a count, as did the DCW. The command indicates the action to be performed, the address the head of a list of DCW's, and the count the length of that list.

To perform a typical I/O operation a user process would first prepare a list of DCW's specifying the operation, and a single LCW directing the associated list channel to feed the DCW's to the desired data channel. It would then activate the list channel, first placing the LCW in its prespecified location. When the list channel has completed its instruction, implying that the list of DCW's was completed by the data channel, it would inform the activator that it was done.

This suggests that a mechanism is needed to activate list channels, and to inform the activators when list channels have completed their assigned tasks. The first is called a control channel, and controls the transfer of the activation command to list channels. The single control channel in each IOC is activated by the interrupt present signal from the controlled memory module. When activated it retrieves a single instruction from a prespecified memory location, called a control channel word (CCW), that specifies the list channel to be activated. To enable coordinated use of the control channel, it sets this instruction word to zero to indicate completion of each task. The second is called a status channel, and controls the transfer of list channel status information from the IOC to memory. The single status channel in each IOC is started each time a list channel completes an instruction. It stores into memory a status word indicating the completed channel. The status word is stored at the location specified by the status channel word (SCW) which is retrieved by the status channel from a prespecified location. In connection with the storing of status the status channel also sends an interrupt to the memory module specified by switch setting in the IOC.

## Mailboxes

The prespecified location from which each channel in an IOC retrieves its instruction word is called its mailbox. The mailboxes for all the channels in an IOC are located in the mailbox segment for that IOC which is part of the IOC data process. The segment number of this mailbox segment is specified by switch setting in the IOC. The channels are numbered sequentially from zero, and the channel number indicates the offset to the associated mailbox word. The mailboxes for the control and status channel, and the data channels are loaded by the action of processes using the IOC to do input/output. The mailboxes for data channels are loaded by the operation of the paired list channels.

## Implementation

In their cycle of operation, the channels of all four types are required to reference memory. As mentioned before, this is done via address mapping logic like that contained in the processors, and the ports and major module interface. Each IOC contains only one copy of the address mapping logic, and all channels of the IOC must share it. Additionally, because the transmission rate of I/O devices is measured in milliseconds per item, and the speed of hardware logic is measured in microseconds per operation, it is possible to share the logic of a data channel among all data channels in an IOC, and the logic of a list channel among all list channels. This substantially reduces the cost of an IOC by making the additional hardware necessary for each list and data channel minimal.

As minimum equipment, then, each channel of any type contains a service request flip flop. When set it indicates to the central unit of the IOC that the channel requires service. The number of the channel implies the type, and this dictates the type of service provided. The actual logic for the channel is part of the central unit. In effect, the logic is time shared by hardware among all the channels of the IOC.

## Identification

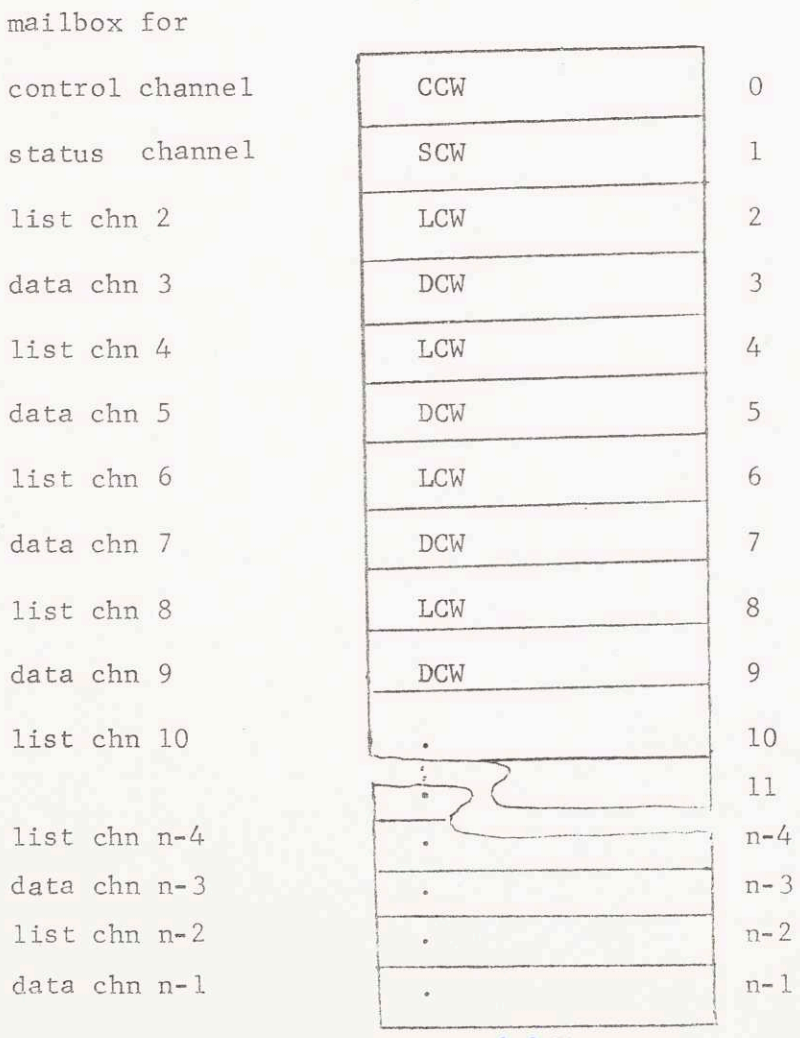
Input/output controller mailbox segment

## Purpose

As described in the previous section, the I/O controllers operate within the address space of the IOC data process. This process provides segments that contain sources and targets for data transfers to and from I/O devices. It also contains a mailbox segment for each IOC. The mailbox segment contains a one word mailbox for each channel in an IOC. The offset to the mailbox for a channel is the same as the channel number.

## Discussion

Following is an illustration of a typical mailbox segment for an IOC containing "n" channels. The control channel is always numbered zero, and the status channel one. List channels are given the even numbers, beginning with two; and the paired data channels the adjacent odd numbers.



Date: DEC 16 1968

## Identification

I/O controller control words

## Purpose

The I/O controllers are directed in their operation by four kinds of control words: control channel words (CCW), status channel words (SCW), list channel words (LCW), and data channel words (DCW). This section describes the format and contents of each of these control words.

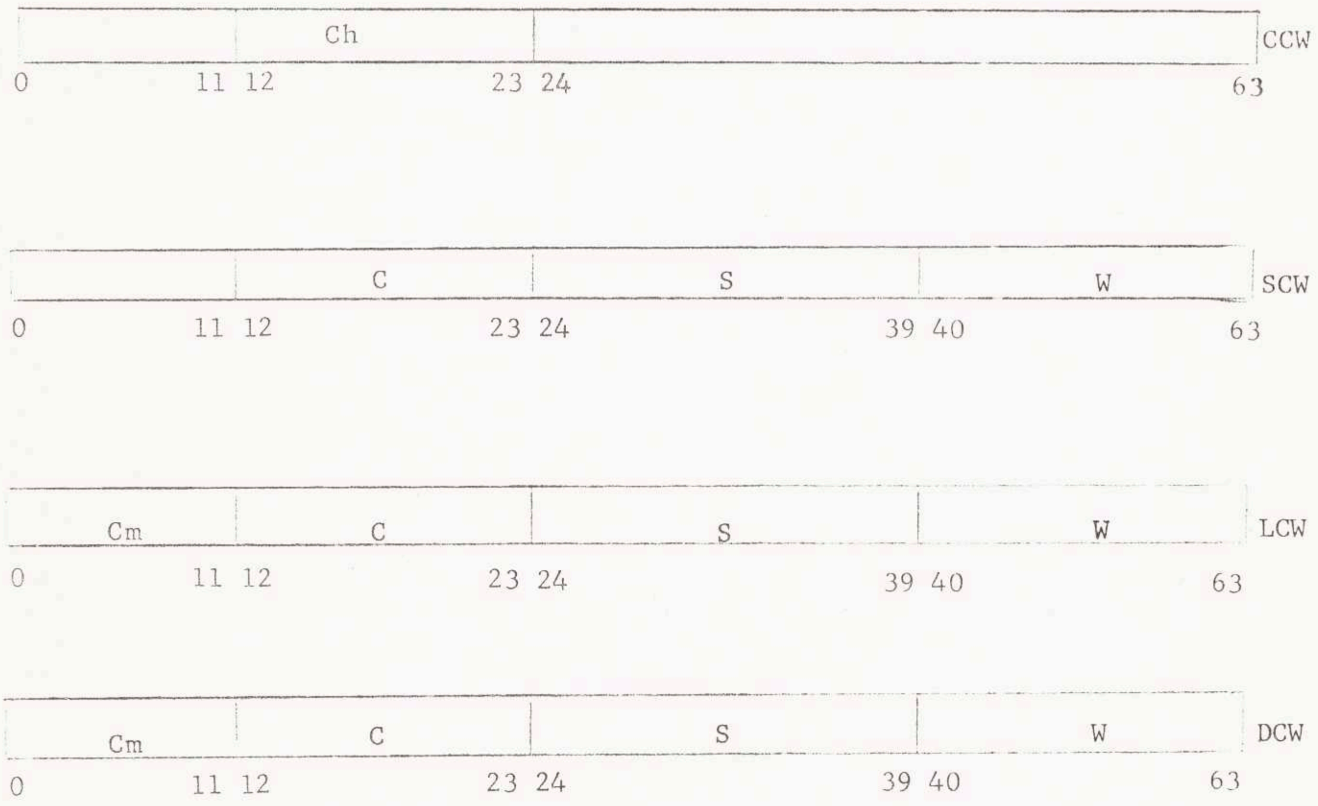
## Discussion

Three of the above named control words contains a segment number and word number address. This address is always interpreted by the I/O controllers as being within the I/O controller data process whose descriptor segment is pointed to by the descriptor base register in the I/O controller. Thus, when forming these control words from a procedure within another process, it is important to translate the segment numbers of data channel word lists and data buffer areas to the number valid in the I/O controller data process.

The diagram on the following page presents the format for each of the four kinds of control words, and gives the field designators. Following is a discussion of the meaning and use of the control words and their fields.

The control channel word is loaded into the control channel mailbox in the mailbox segment of an I/O controller by a processor. Each CCW contains the channel number of a list channel. When the I/O controller acts upon the CCW the service request switch for that list channel is then set on, and the entire CCW set to zero. The CCW becoming all zero is the signal to the processor that it has been acted upon.

Status channel words are loaded into the status channel mailbox of each I/O controller. The SCW points by segment number and word number to the head of the status information list .It also contains the length of the list . Each time the I/O controller stores status, it is stored in the location indicated by the segment and word number in the SCW. The word number is then incremented by one and the count decremented by one. When the count becomes zero, the I/O controller sends a trouble interrupt. It is the responsibility of the processors to update the SCW as the status words stored in the status information list are noted.



| field | use |
|---|---|
| C | count |
| Ch | channel number |
| Cm | command |
| S | segment number |
| W | word number |

<u>I/O Controller control words</u>

The list channel word is placed in the mailbox for a list channel.
The LCW controls the retrieval of DCW's for the associated data channel. It contains the segment and word number of the head of the list of DCW's, and the number of words in the list. Each time a DCW is retrieved and placed in the mailbox of the data channel the word number is incremented and the count decremented by one. When the count becomes zero, completion is signaled via the status channel. The LCW also contains a command. This allows the user to specify certain options affecting the operation of the list channel. The list channel commands are not contained in this version of the Clics notebook.

The data channel words are retrieved from a user provided list and placed in a data channel mailbox by the list channels. Each DCW controls directly the operation of an I/O device. It contains a segment number and word number indicating the source or target location for data transfers, a count indicating the amount of data to be transfered, and a command indicating the operations to be performed. When a DCW is exhausted, control is passed back to the associated list channel to retrieve the next DCW, or to signal completion. The DCW commands are not specified in this version of the Clics notebook.

Date: DEC 16 1968

## Identification

Overview of memory module organization

## Purpose

The memory modules of the Clics system are divided into three functional units: the system controller, the clock driver (optional), and the storage unit. This section describes in general terms the purpose of each unit, and indicates their interaction.

## Introduction

The memory modules are the only type of passive module in the Clics system. Passive modules cannot perform instructions, but can only respond to externally supplied commands. These commands are received from the up to 16 instances of the major module interface that are connected to the 16 ports of each memory module. Each active module in the system (processor or I/O controller) may request that a command be accepted by any memory module in the system. The accepting unit is the system controller, which performs the command itself, or passes it on to the storage unit. Acceptance of commands is controlled by a prewired priority scheme in which each port on each memory module has a fixed priority in relation to the other ports on the same module. When simultaneous requests are received, the request from the port with the highest priority is accepted first.

One port on each memory module is designated by switch setting as the control port for that module. The processor or I/O controller connected to that port is the control module for the memory. The control module has special privileges and is the recipient of all intermodule control communication directed through the memory module.

## System controller

The system controller is the controlling unit of each memory module. Its responsibility is to accept by priority the requests of the connected active modules, decode them, and direct their performance. In some cases, the system controller may perform the command unassisted, while in others the storage unit assists. The interrupt cell is part of the system controller.

## Clock driver(optional)

The clock driver portion of each memory module is optional. Normally only one or two of the memory modules in a system will contain clocks. In a memory module that does not contain a clock the clock driver unit is missing. If a clock is present in a memory, two words in storage are designated by switch setting in the clock driver as the calendar clock and the alarm.

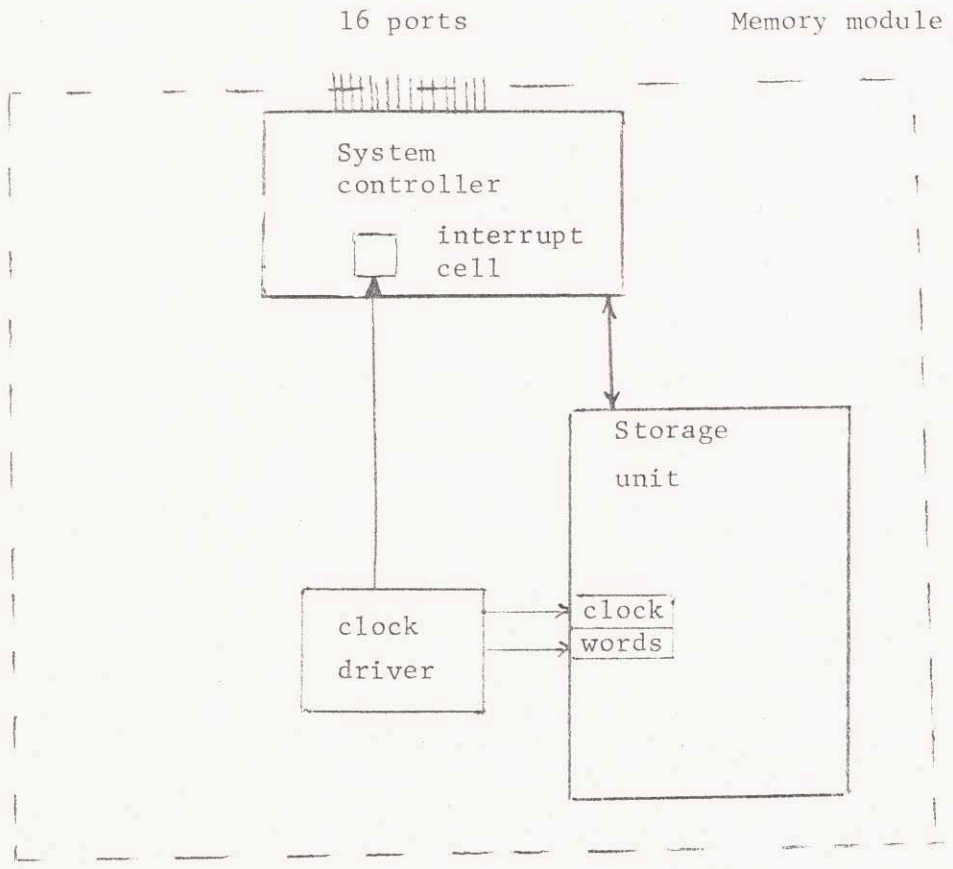
The former is incremented once each $\mu$ second by the driver. At each cycle the latter is compared with the calendar clock. When they become equal the interrupt cell in the system controller portion is set on, causing an interrupt present signal to be sent to the controlling active module. Both clock words may be read and written as memory locations.

## Storage unit

The storage unit in each memory module contains 67,108,864 64 bits words. The access time to any word is approximately 1 microsecond. Two types of access are permitted: read/restore and clear/write.

## Diagram

The following diagram illustrates the memory module organization



## Identification

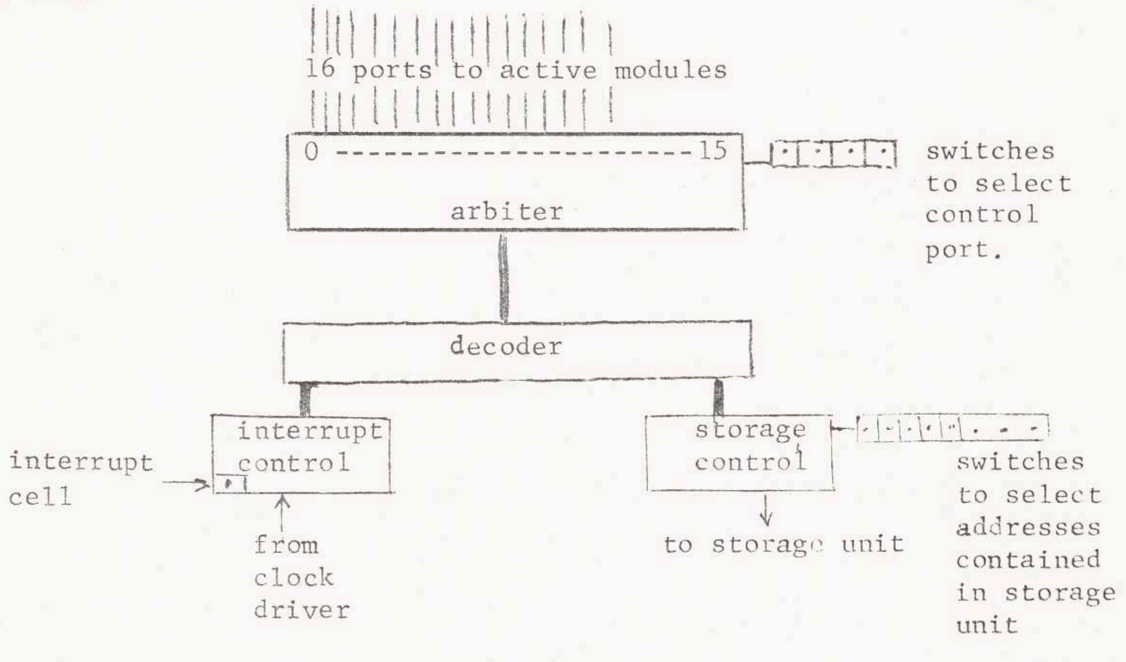
System controller

## Purpose

The system controller is the controlling logic unit of each memory module. It intercepts and decodes service requests from the active modules of the hardware system, and directs the performance of the requested service. It also contains the interrupt cell that is part of the interrupt communication hardware. This section describes the organization and function of the system controller.

## Introduction

The system controller is divided into four units: the arbiter, the decoder, storage control, and interrupt control, The following diagram illustrates these components and their relationship to each other



To each of the 16 ports on the arbiter may be connected an active module. The connection in each case is via the major module interface. The arbiter selects which active module service request to honor, and passes the input information to the decoder. This unit interprets the request and determines which of the two control units should process it. The input information is then switched to the proper unit.

The communication lines between the arbiter and the decoder; and between the decoder and the control units are copies of the major module interface, except that the interrupt present lines are missing. Thus, the decoder has the entire input available from the port selected by the arbiter, as does the relevant control unit. There are additional communication paths among the four units of the system controller, two of which are functionally interesting. The interrupt present line from the control port for the memory module (the control port is specified by four binary switches on the arbiter) is connected directly to the interrupt control unit, allowing that unit to signal the fact that the interrupt cell is set to the control module. A second line, from the arbiter to the interrupt control unit, allows the former to indicate to the latter that the service request is from the control module. This is necessary because the request to execute the interrupt is only honored when made by the control module.

Following is a description of each of the units in the system controller.

## Arbiter

The arbiter is directly connected to the 16 ports of the memory module. Each of these ports can be connected via the major module interface to an active module in the hardware system. One of the ports is designated by switch setting as the control port for the memory module, and the connected active module is the control module.

The arbiter's cycle of operation begins with a scan for access requests. This is accomplished by simultaneously setting the 16 bits of a special arbiter register from the access request lines of the 16 memory module ports. A combinational circuit then selects the register cell that is on. If more than one is on, the cell associated with the lowest port is selected. Thus, selection is on a first come, first served basis except when simultaneous requests are made. If no register cell was on the action is repeated. This is continued until a signal is found, indicating an access request from the connected active module. 112 lines of the major module interface are then switched to the input/output bus for the decoder unit. (The 112th, the interrupt present line, does not participate in this action.) The arbiter then waits for the access request signal to be removed, indicating that the requesting active module has received the output associated with its request. The lines are disconnected from the decoder bus, and the cycle of operation begins again.

A variation of this cycle will occur if the command received from the active module is RAH (read and hold). The recognition of this command on the command lines by the arbiter causes it to enter the hold mode. In this mode the removal of the access request line signal by the active module does not cause the arbiter to recycle. Rather, service is granted to the same active module again, allowing it two sequential accesses to the memory module. Following the second access the cycle terminates in the normal manner, and begins again. The ability to obtain two consecutive accesses to a memory module is used by the processors to implement the "STOZ" and the "ADDS" instructions. (See the description of these in the instruction set section.)

## Decoder

The decoder function as a two way switch. When signals appear on its input/output bus from the arbiter it examines the command code and determines which of the control units to activate. The following table lists the possible commands and indicates the activated control unit.

| command to memory module | system controller control unit |
|---|---|
| READ | storage control unit |
| WRITE | " |
| RAH | " |
| SIC | interrupt control unit |
| EXI | " |

The 111 lines of input/output information are switched to the input/output bus for the appropriate control unit. When the signal disappears from the access request line the connection to the control unit is broken.

## Storage control unit

The storage control unit interfaces with the storage unit of the memory module. If the input command is READ or RAH then the storage unit is given the command READ/RESTORE. If the input command is WRITE then the command CLEAR/WRITE is given the storage unit. Parity errors and unknown addresses are reflected back to the requesting active module as final status codes. The output information is kept on the output lines until the access request line signal is removed. The addresses in the storage unit are set by switch on the storage control unit.

## Interrupt control unit

The interrupt control unit performs the two commands associated with interrupt communication. In response to an SIC (set interrupt cell) command the interrupt cell in the unit is set on. The EXI (execute interrupt) command may only be performed at the request of the control module for a memory module. Thus, before it is performed the special communication line from the arbiter is checked for a signal indicating the request was received via the control port. If not a final status of "illegal operation was requested" is returned. Assuming the request is valid, the interrupt cell is set "off".

The interrupt control unit also keeps constant watch of the interrupt cell and sets a signal on the interrupt present line connected to the control port whenever it is "on".

There is a special interface between this unit and the optional clock driver unit that allows the latter to set the interrupt cell when the alarm "rings".

Date: DEC 16 1968

## Identification

Storage unit

## Purpose

The storage unit in each memory module contains 67,108,864 words of absolutely addressable 64 bit words. The words are contiguously addressed from a base address that is set by switch on the storage unit. The operations READ/RESTORE and CLEAR/WRITE are permitted. Error codes are returned for parity and unknown address. If the memory module contains a clock then two words in the unit are designated the calendar clock and the alarm. (See clock driver notebook section.)

Identification

Clock driver unit

Purpose

If a memory module contains a clock, then it includes a clock driver unit. The driver has direct access to the calendar clock and alarm words. It increments the former by one each microsecond. When the former becomes equal to the latter is signals the interrupt control unit in the system controller to set the interrupt cell "on". The clock driver is informed of the addresses of the clock words by switch setting.

Date: DEC 16 1968

## Identification

Tape handler overview

## Purpose

Tape handlers are one of the two types of I/O devices in the Clics system. The specification of these devices is not contained in this version of the Clics notebook.

Date: DEC 16 1968

## Identification

Overview of typewrite terminals

## Purpose

Typewriter terminals are one of the two types of I/O devices in the Clics system. The specification of these terminals is not contained in this version of the Clics notebook.

Date: DEC 16 1968

## Identification

Storage management subsystem overview

## Purpose

The storage management subsystem (SMS) controls the allocation of physical memory and maintains the structure of stored information generated by the Clics community of users. The design of the SMS reflects the assumption that each user of Clics must have the ability to create, name, manipulate, reference, and destroy arbitrarily sized units of information, called segments, within his own environment; and further, that each user must have the ability to share these segments in a controlled manner with other users of the Clics system.

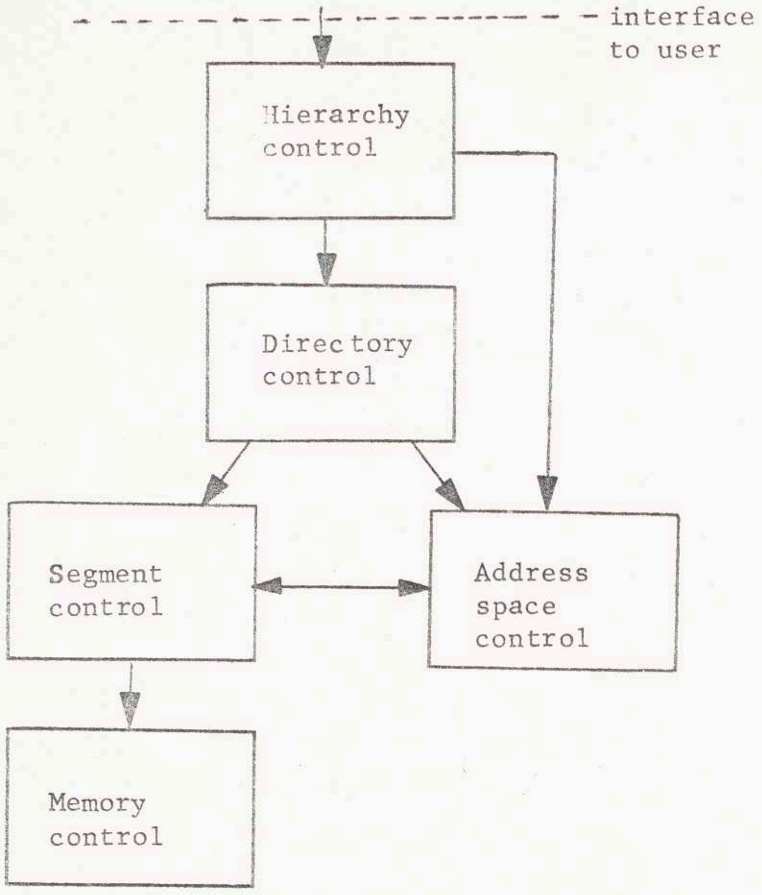
Clics itself makes extensive use of the facilities provided by the SMS, storing all control programs and system data bases in its information structure. In fact, with the single exception of the list of unused memory blocks, no information exists within the Clics system that is not a uniform part of the SMS information structure.

A user avails himself of the facilities of the SMS by logging-in to the Clics system and gaining control of a process.Procedures are then activated which call the entry points in the SMS on behalf of the user. In the following descriptions the informality of speaking about the user of an SMS module is frequently seen. Who (or what) is meant is usually clear from context, but it must be kept in mind that the procedures that comprise the SMS (or any other Clics subsystem) are activated only by calls from other procedures, and not directly by a user seated at a typewriter terminal. The most direct activation possible is for a user to instruct the command subsystem to construct and execute a call directly to an SMS entry point.

## Introduction

The SMS is divided into five groups of modules, which are named memory contol, segment control, address space control, directory control, and hierarchy control. Each group may be viewed from the outside as a block box performing certain well defined services when activated by calls to its interfacing entry points.

Following is a description of each group from this point of view. Later sections detail the structure of modules within each group. The diagram following represents the gross relationships. Arrows represent calls between groups.



Memory control

Memory control is the most primitive area of the SMS, interfacing with physical memory and shielding the other SMS groups from the inhospitable environment of the raw hardware. When viewed at its upper interface its three entries are seen to provide the ability to create segments, change their length, and destroy them. The only attribute that segments have at this level are an absolute base address and a length in words. A call to create a segment of a specified length is processed by constructing an appropriately connected structure of pages and page maps from free blocks of memory, and returning its absolute base address. Alteration to the length of a segment identified by base address (and old length) involves restructuring the pages and page maps, adding or returning memory blocks as necessary. Destruction of a segment results in all blocks occupied being returned to the pool of free memory.

Segment control

Segment control is the next layer of the SMS: It calls upon memory control to perform the three functions described above, and presents callable interfaces to both directory control and address space control. The primary data base in segment control is a system-wide segment catalog, which is organized as a linear array, and contains an entry for each segment within the system. Requests are received from directory control to create, alter, delete, and list the attributes of segments. Memory control is invoked, in turn, to perform the implied physical manipulations to segments. When a new segment is created a catalog entry is made for it, and a unique segment identifier assigned.

The length supplied, and the absolute base address returned by memory control are included in the catalog entry. An empty access control list is also stored there. The assigned segment identifier is returned to the creator. The segment's length can later be altered and access control list grown and shrunk by calling segment control with the identifier, and the proper alteration specifications. The information from a catalog entry may be listed, and the entry and the associated segment may be deleted

In addition to cataloging segments in the manner described above, segment control allows segments to be referenced (read, written in, or called). When presented with a segment identifier by address space control it will return information from which a segment descriptor word (SDW) for the segment can be built. When added to a process'descriptor segment, this SDW allows procedures within the process to reference the segment by segment number. The processes which can be given a SDW for a particular segment are defined by the access control list in the catalog entry. This list specifies Clics users which may cause their processes to reference the segment, and specifies the access mode indicator that must appear in the SDW given to each such process, thus controlling the type of access permitted. If a process whose owner does not appear on a segment's access control list asks segment control for a SDW for the segment, segment control will not provide it, denying the process access to the segment.

Each time a process is given a SDW allowing access, record of the action is made in the segment's catalog entry. At any time, then, a list of all processes using a segment is available. (The record includes the image of the descriptor base register of each process, and the segment number assigned.) Use of this list is made in two instances. First of all, if a segment is deleted while being used by one or more processes the fault bit in the SDW's of each process described on the list is set "on",preventing the process from further accessing the deleted segment. The second instance occurs when the alteration of a segment's length causes its absolute base address to be changed. The fault bits in all using process' SDW's are again set "on". (Address space control is called to actually set the bit.) The set fault bit means that the next reference to the segment by each of the processes will cause a descriptor segment fault to occur.

It handling the fault each process will attempt to re-include the segment in its address space. If the segment has been deleted this attempt will of course fail. If only its base address has changed, a new SDW with the correct base address is supplied. When a process determines that it no longer needs to reference a segment it informs segment control so that the use record of the segment may be updated.

## Address space control

This group appears at the same level of the SMS as segment control, and performs the initialization of a process'address space that is necessary to allow procedures in the process to reference a particular segment. The segment that a process wishes to reference is specified to address space control by directory control with an identifier. If a SDW for the segment already exists in the process' descriptor segment, then the segment is already referenceable, and a pointer to its zeroth word is constructed from its segment number and returned. If no SDW exists then segment control is called to provide one. If it does so, then the SDW is included in the descriptor segment, and a pointer constructed and returned. In the event that the segment is not referenceable by the process a null pointer is returned instead.

A special entry to address space control allows the other modules in the ring zero system to cause all segments included in a process' address space after it became assigned to its current owner (a system user) to be removed.

Address space control is also responsible for processing segment descriptor faults. Recall that these could be set by segment control to indicate that an SDW in the process' descriptor segment contains invalid information. (The fault bits in descriptor segment words that are not in use are also kept "on".) The reason the bit triggering a specific instance of this fault was set is determined, and if set by segment control, then segment control is called to obtain a new SDW which can be used to reset the fault. If no new SDW is forthcomming then the fault cannot be corrected,and control is forced back to the listener by the processor management subsystem.

## The SMS as described to this point

At this point it is profitable to consider what kind of a storage management subsystem is provided by just the three groups of modules described so far. Users may cause their processes to create and manipulate segments specified by system generated unique identifiers. Processes may include these segments in their address spaces and reference them, subject to the constraints of each segment's access control list. This list may be altered by calling segment control. If the growing of a segment alters its absolute base address (and the number of layers of page maps), or a segment is deleted, processes using the segment are immediately informed via the SDW fault bit.

Clearly, this does not provide the general mechanism required to control sharing of segments among user's processes, for each process has the ability to arbitrarily change the length and access control lists of, and destroy segments. It also does not provide users with the facilities for naming and grouping segments,i.e. imposing a content related organization on the segments within the system. It does provide, however, the foundation for the general mechanism. The remaining two groups in the SMS, when placed on top of this foundation, complete the mechanism.

Before considering directory control and hierarchy control it will be helpful to consider the data structure around which these groups are built. In order to control the ability to manipulate the system segment catalog, and to allow users to impose a content related organization upon the catalog, a hierarchical structure is placed on top of that catalog. This hierarchy is now described.

## The SMS hierarchy

The SMS hierarchy (whose fundamental purpose is to name segments and organize them into groups) is formed from segments called directories. Directories are ordinary segments when viewed at the level of segment control, and are described by entries in the system segment catalog. Above this level, however, both in the SMS and outside of it, directories are treated in a special way. Only the procedures of the SMS (directory control, in particular) may access and manipulate directories directly. Each directory contains a list of entries, called branches, which describe segments in the segment catalog. The following list is indicative of the information contained in a branch:

branch name
segment identifier
directory switch
(directory control list)

Each branch describes a single segment in the system segment catalog. It contains a branch name for the segment, the segment identifier of the segment in the catalog, and a switch specifying whether the segment is another directory segment or not. If it is a directory then the branch also contains a directory control list which will be described later. (Directory segments also have access control lists in their system segment catalog entries.)

If the directory switch is "on" then a branch describes another directory segment. Since closed loops are prohibited, directories containing branches describing directories, etc., form a tree or hierarchy. This tree is begun with a special directory with the branch name "root". Since there can be no directory containing a branch describing "root" each process is directly given with access to this special directory by making it part of its address space and supplying it with a pointer to a "root" before it begins executing. In addition to "root" the Clics system provides other directories in the initial portion of the tree. (See the notebook section on the hierarchy organization.)

Any branch in the hierarchy may be uniquely specified by its tree name. A tree name is formed from the branch names of the directory segments along the path from "root" to the directory containing the branch, and the branch name in the branch itself. (Frequently the phrase"tree name of a segment" is used. This means the tree name of the branch describing the segment.) These branch names, called components of the tree name, are listed left to right and separated by periods in their normal representation. "root.pdirdir.pl.temp" is an example of a tree name with four components. If the branch thus identified exists within the hierarchy, it will contain the branch name "temp"and appear in the directory whose branch with branch name "pl" will appear in the directory whose branch with branch name "pdirdir" will appear in the "root" directory. The directory segments whose tree names are formed by any contiguous left group of components of a tree name, beginning with "root", are said to be superior to the segment named by the tree name. Thus, the directory segments named (whose branches are named) by "root", "root.pdirdir" and "root.pdirdir.pl" are superior to the segment named in the above example. The last of these is immediately superior. The reverse relationships are inferior and immediately inferior. A segment may have only one immediately superior directory, but a directory may have many immediately inferior segments. (Segment is used to denote either a directory segment of a non-directory segment. When only one or the other is meant, directory or non-directory is used.)

If a branch describes a directory segment, then the branch also contains a directory control list. A directory control list is somewhat analogous to the access control list that appears in system segment catalog entries of all segments in that it also consists of a list of Clics users. In this case the list is of those on whose behalf the SMS will manipulate the directory. Since segment catalog entries can be considered as extensions of the corresponding directory branches, permission to cause catalog entries to be altered is keyed to permission to cause the directory containing the branches to be altered. Users not specified on the directory control list in a branch for a directory segment may not cause the described directory segment, or contained branches and the corresponding catalog entries, to be altered.

In order for the segment manipulation constraints and content related organization supplied by the hierarchy to be completely effective, users of the SMS must be forced to interface with segment control and address space control through the hierarchy. Thus, user procedures are not allowed to call the previously described portions of the SMS directly. Instead, they must call hierarchy control, which with its servant, directory control, imposes the organization and constraints of the hierarchy upon user requests. These two SMS groups are now described.

## Directory control

Directory control is positioned immediately above segment control and address space control, and invokes both in the performance of requests received from hierarchy control. Directory control is not "aware" of the SMS hierarchy. It deals with isolated directories through pointers provided by hierarchy control. Its environment of operation is the directory, the contained branches, and their contents. Requests implying action on a lower level of the SMS are passed-on to segment control and/or address space control.

Two groups of requests are accepted. The first includes the following seven items:

1. create a branch and segment
2. change a non-directory segment's length
3. delete a branch and segment
4. rename a branch
5. update a control list (access or directory)
6. list a branch and its catalog entry's contents
7. list branch names in directory

In all cases the names of the branch or branches involved are supplied along with a pointer to the relevant directory. Request 1 causes a new branch to be created with the given name, and segment control to be invoked to produce the corresponding segment. Request 2 causes the length of the non-directory segment corresponding to the named branch to be altered by invoking segment control. (Note that the identifier required by segment control in this and the following cases is obtained from the named branch, where it was placed when the branch and segment were created.) Request 3 results in the removal of the named branch from the directory and destruction of the corresponding segment by segment control. Request 4 can be processed completely within directory control and involves only changing the name in a branch. Request 5 produces a different result depending on whether the named branch describes a directory or a non-directory segment. In the former case the directory control list in the branch is updated as specified, while in the latter the access control list in the corresponding catalog entry is altered by invoking segment control. Requests 6 and 7 involve listing selected information from the directory, its branches, and the corresponding system segment catalog entries.

In each of the above cases the permission of the user making the request to cause the associated manipulation to occur is validated by hierarchy control before directory control receives the request. The relevant directory control list is that for the directory containing the branches being manipulated, and is found in the branch for that directory in the immediately superior directory.

The second group of requests processed by directory control is much shorter. Included are the following two:

1. obtain pointer to non-directory segment whose branch name is given.
2. obtain pointer and relevant directory control list entry to named directory segment whose branch name is given.

Filling these requests involves locating the named branch (if it exists) and then invoking address space control to obtain a pointer to the corresponding segment. In the second the applicable directory control list entry from the branch is returned with the pointer. No validation of this request is necessary within directory control.

## Hierarchy control

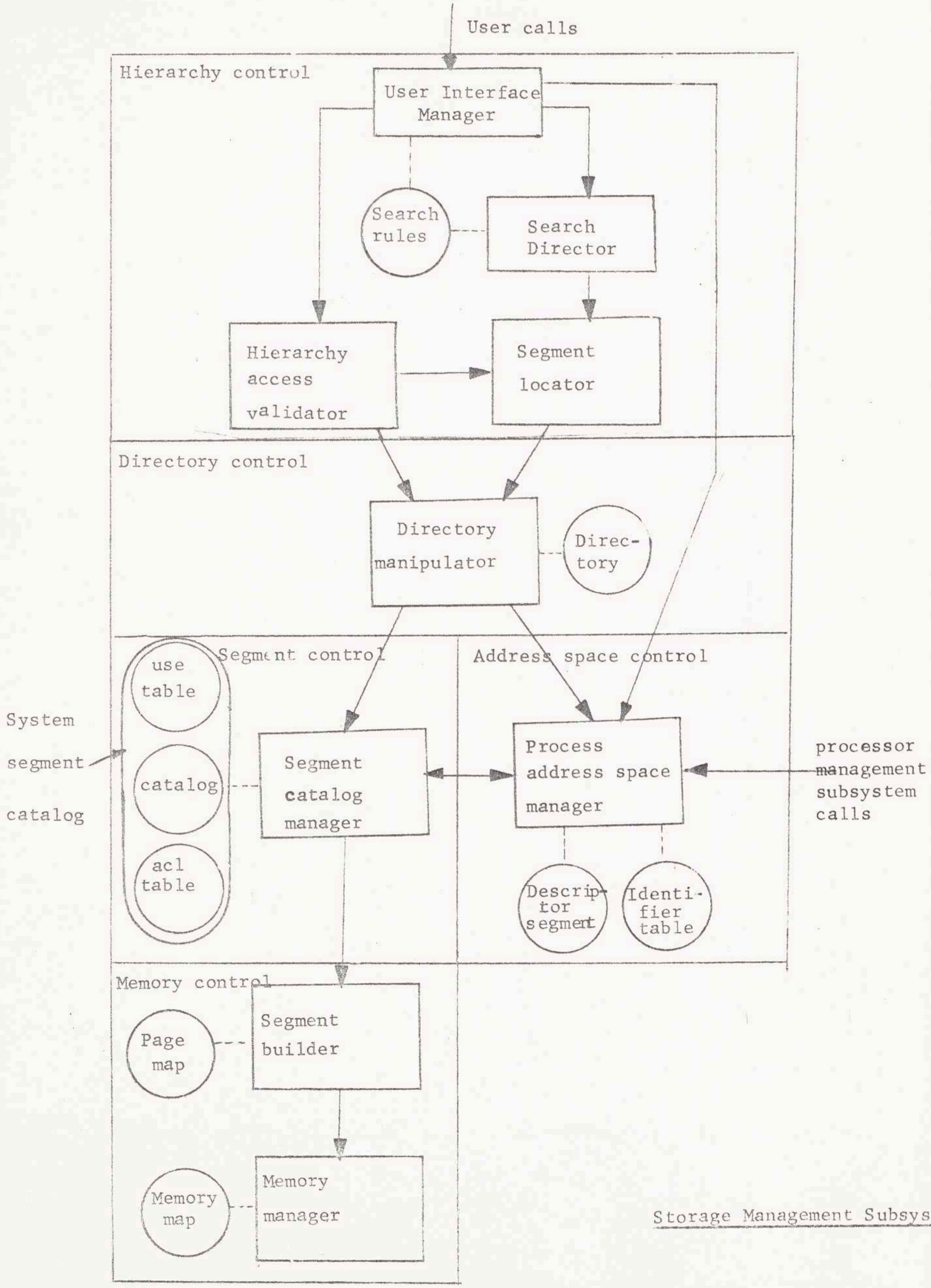
Hierarchy control is the top-most layer of the SMS, and, as such, interfaces directly with requests from user procedures. It is the only portion of the SMS "aware" of the hierarchy of directories, and is responsible for finding the proper directory and validating the requested manipulation before passing the request on to directory control. In these tasks it uses the facilities of directory control.

Both groups of requests accepted by directory control are represented in user available entries to hierarchy control. Those in the first group are accompanied by the partial or full tree name of the containing directory as well as the branch name. The tree name is completed, if necessary, and the hierarchy is searched for this directory, using directory control to provide the pointers to directories along the path from "root" to the directory specified. The request is matched against the directory control list of the directory, and directory control invoked to perform it only when a pointer to the directory is available and the request is properly validated.

The second group is represented by a single entry allowing user procedures (and the linker) to obtain pointers to non-directory segments. Again, the segment desired is identified by partial or full tree name. The pointer, of course, will only be returned if the segment specified exists and the owner of the requesting process is on its access control list. It is never returned if the named segment is a directory, for user procedures are not allowed to access directory segments.

SMS module diagram

On the following page is a diagram including all of the procedures and data bases of the SMS. Rectangular boxes represent procedure segments, while circles represent data segments. Calls between procedures are represented by arrows and data base references by dashed lines. All SMS procedures and data segments reside in ring 0, and may be called only from within ring 0. Entry to the SMS for user procedures is via the ring 0 entry vector.



Storage Management Subsystem

## Identification

Sharing and protection of segments

## Purpose

To control the sharing of segments access control lists are placed in system segment catalog entries. To control the manipulation of directories, directory control lists are placed in branches describing directory segments. This notebook section describes each type of list, and the way in which each is interpreted by the modules of the SMS.

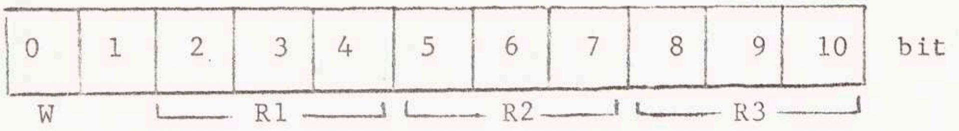
## Introduction

There are two aspects to the control of segment sharing. The first is control of who may cause their processes to read, write, or call each segment. The second is control of who may manipulate directories. Access and directory control lists are the source of the control parameters in each case, respectively. Each instance of each type of list is an array of entries containing a name and an indicator. The name identifies a particular user of the Clics system, and can be drawn from the set of unique identifiers specifying members of the Clics community of users. It can also be the character "*", signifying the set of all users in that community.

Each time a user logs-in to the system and gains control of a process (becomes its owner) his identifier is recorded in the process catalog entry of the process. Thus, a process may determine the identity of its owner simply by referencing that data item. When the SMS is called by a procedure in the process, it executes as part of the calling process. When the point is reached in the SMS procedure that an access or directory control list must be interrogated, the SMS retrieves the name of the owner of the process in which it is executing. This is the user on whose behalf the request is being made. The relevant entry on the list being considered is then selected by searching the list from top to bottom and accepting the first entry containing a matching name. The name "*" is considered to match all user names. If no match is made in the list, the user is considered to have no permission. If a match is made, then the indicator in the entry specifies the user's permission to perform manipulations and references upon the directory or segment. The specific interpretation of the indicator differs for access and directory control lists. Access control lists are considered first.

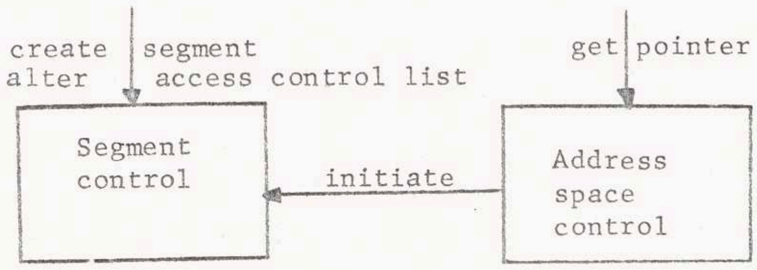
## Access control lists

An access control list appears in the system segment catalog entry of each segment within the Clics system. Its contents are specified (and may be updated) by any user having manipulation permission in the directory containing the branch for the segment (explained later). The indicator in each entry is called an access mode indicator, and has the following format:



It is exactly a hardware access mode indicator of the kind expected by the processor address mapping logic in segment descriptor words (SDWs). The interpretation of such an indicator by the hardware is described in the notebook section on segment access protection in the processors. Generally speaking, it limits the rings from which a segment can be read, written, or called; and specifies whether it can be written at all. Its constraints are enforced by the hardware at each reference to a segment by a process.

The access control list of a segment is used to provide SDW access mode indicators to processes wishing to reference the segment. The relevant manipulations occur at the segment control and address space control levels of the SMS. The following diagram illustrates the basic interaction:



A call to the create entry in segment control from directory control creates a new segment. The assigned segment identifier is returned. The initial access control list is then provided by a call to the alter access control list entry point. A call from a procedure wishing to reference the segment eventually reaches the get pointer entrance to address space control. The segment to be referenced (specified by segment identifier) and the ring occupied by the calling procedure are given as arguments. Address space control calls the initiate entry in segment control with the segment identifier and ring number. Segment control attempts to locate the corresponding system segment catalog entry. (If unsuccessful in this operation a not-found flag is returned.) If successful, then the segment's access control list is searched. If no matching entry is found a not-found flag is again returned. Otherwise, the access mode indicator from the matching entry is checked to verify that at least one of R1, R2 or R3 contained therein is greater than or equal to the caller's ring, and then returned to address space control along with the absolute base address and number of levels of page maps for the segment. From this information address space control constructs and inserts into the descriptor segment of the process an SDW, thus allowing the segment to be referenced. Thus, the access control list in the catalog entry controls the reference by

i) specifying whose processes might include the segment in their address spaces and
ii) specifying the access mode indicator to be used by each that may.

## Locating a segment

That described above is the final step in locating a segment to be referenced. In order to get as far as that step, however, the segment identifier of the segment desired must be determined from the segment's tree name. This process may be initiated for two reasons: to locate a non-directory segment for reference by an arbitrary procedure in a process, and to locate a directory segment for reference by the SMS. (If a user procedure provided tree name identifies a directory segment, a pointer to it is never returned. Only the SMS may reference directories.)

A module within hierarchy control named the segment locator controls segment location. It receives the full tree name of the segment to be located. The first step is to obtain a pointer to the "root" directory.

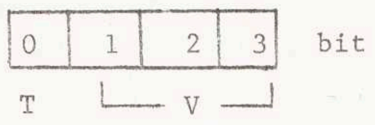
(If root is the only component in the tree name given, then the task is completed. The pointer is returned only if the call is from the SMS itself.) The second component in the tree name is the branch name of the next directory along the path to the desired segment. This branch is in "root". Directory control is called with the pointer to "root", and the branch name. The request is to obtain a pointer to the directory described by the branch. This is accomplished by directory control by searching "root" for the named branch. If found, then the segment identifier in the branch is passed on to address space control to obtain a pointer to that directory. A pointer will always be returned for the access control list for directory segments, by system convention, allows all users read and write permission in ring zero. Directory control then returns this pointer back to the segment locator.

The segment locator now has a pointer to the second directory in the path. The method used to obtain it can be repeated to obtain a pointer to each remaining superior directory to the segment desired, in succession. If this process is successful, i.e. if all these directories exist in the hierarchy, the segment locator will end up with a pointer to the immediately superior directory to the segment desired. The final component of the tree name given will be the branch name of the segment within that directory.

If the call to obtain the segment is from the SMS itself, the process is repeated one last time, and the resulting pointer returned as that to the desired directory. If the call to obtain the segment is from a user procedure, then the last step is different in a small detail. Directory control is called to obtain a pointer to a non-directory segment. The pointer to the directory to be searched, and the branch name are given. A null pointer is returned if the named branch cannot be found, or if the branch describes another directory. Otherwise, address space control is passed the segment identifier contained in the branch, and returns a pointer to the segment it the owner of the process is on its access control list. This pointer is returned back to the segment locator, and then back to the initial caller.

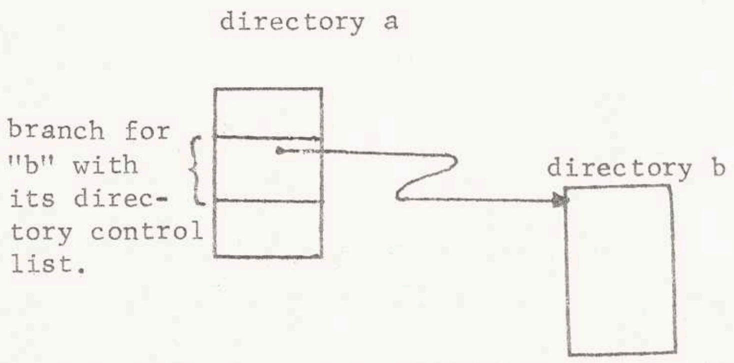
## Directory control lists

A directory control list appears in the branch describing each directory segment. (Directory segments also have access control lists in their catalog entries giving all users read and write permission from ring zero.) The indicator in each directory control list entry is called a directory access indicator, and has the following format:



The T bit specifies the type of permission, with T=0 indicating list permission and T=1 indicating manipulation and list permission. V is the validation level. The call to the SMS must have originated from ring V or lower for the permission T to apply.

The directory control list of a directory is in its branch, which is located in the immediately superior directory.



A request to list the names of all branches in directory b, or the contents of any one branch plus the contents of the corresponding catalog entry is performed if a matching directory control list entry in the branch for directory b (located in directory a) is found, and the request comes from ring V or lower. Of course, if no matching entry is found then the request is not honored. A request to manipulate directory b is honored only if the matching entry indicates T=1, and comes from ring V or lower.

As described in the SMS overview, five manipulations and two types of listings may be requested. The permission of a user to cause one of these seven events to occur in a directory is validated by hierarchy control. As described earlier, a module named the segment locator that is part of hierarchy control is responsible for locating the needed directory. Along with a pointer to the directory previously described as being returned, the directory access indicator from the directory's directory control list entry which matches the owner of the calling process is returned. Permission to cause the requested action is validated by matching the request to the indicator. (If no indicator is returned, hierarchy control rejects the request.) Below are listed the seven possible requests, and the condition necessary for the performance of each. If the conditions are met then directory control is invoked to perform the request.

1. create a branch and the associated segment - T = 1 & V ≥ ring occupied by requesting procedure.

2. change a non-directory segment's length - same as 1.

3. delete a branch and the associated segment - same as 1.

4. rename a branch - same as 1.

5. update the access control list of a non-directory or the directory control list of a directory whose branch is in this directory - same as 1.

6. list a branch and the contents of the associated catalog entry- (T = 0 or T = 1) & V ≥ ring occupied by requesting procedure.

7. list all branch names from directory - same as 6.

One further constraint completes the mechanism. When request 5 is performed, the V field in directory control list entries or the R1, R2, and R3 fields in access control list entries being added may never be specified by a requester as being less than the ring from which the request was made. This prevents, among other things, a user from creating a procedure segment that can be called by him and will execute only in ring zero. Note that permission to manipulate the directory control list for directory b (in the previous example) is given by T=1 permission in the directory control list for directory a, which is in the branch describing a in the immediately superior directory to a. (Remember that b's directory control list is in the corresponding branch in directory a.)

Date: DEC 16 1968

## Identification

Hierarchy organization

## Purpose

The Clics system provides a portion of the SMS hierarchy for users to work with. This section describes the provided portion, and documents certain conventions enforced by the system on the use of the hierarchy. Also included for reference purposes is a description of the location of certain data bases and other segments not yet described.

## Basic organization

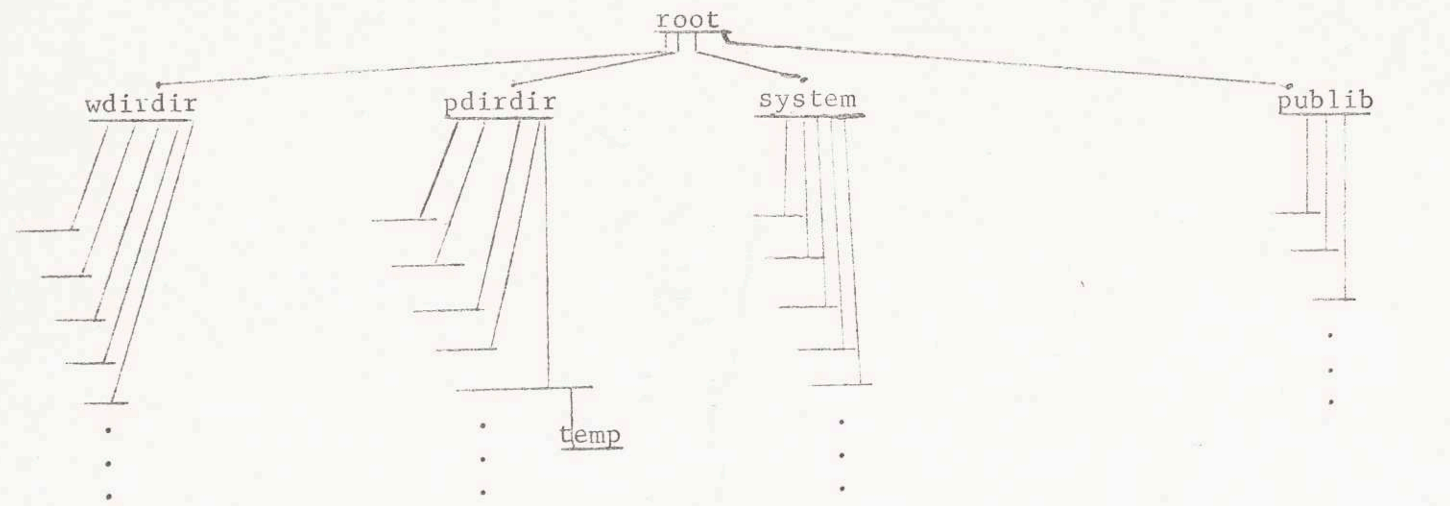
The diagram on the following page summarizes the initial portion of the SMS hierarchy that is provided by the system. The "root" directory contains four branches. The table below summarizes these:

| Name | Use |
|---|---|
| udirdir | directory of user directories, containing a branch describing a single directory for each member of the Clics community of users. |
| pdirdir | directory of process directories, containing a branch describing a single directory for each process in the Clics system. |
| system | system segments directory, containing a branch describing each system-wide data base, and the text and master linkage segments for each system procedure. |
| publib | public library, containing a branch describing the text and master linkage segments for each library procedure. |

The directory control lists for "root", and for each of the four directories described above contain the following entry:

| ADMINISTRATOR | |
|---|---|
| T = 1 | V = 0 |

Thus, only a special system administrator may list and manipulate them.



working directory for each member of the Clics community of users

process directory for each process in the system

all system-wide data bases and control programs

all library procedures

Basic SMS hierarchy structure

## User directories

The user directory for each person in the Clics community of users that is defined by a branch in "root.udirdir" is given the branch name that is the identifier by which the system knows the user. Its directory control list contains a single entry giving the user T = 1 permission from ring 4, the ring in which a user gains control of a process when he logs-in to the system. (A user may force a process into a higher ring by simply resettting the procedure ring register with a larger value.)

## Process directories

The branch name of the process directory corresponding to each process is the process' identifier. The directory control list of a process directory exists in two states. If the process is not owned then it is completely empty. When a user logs-in to the system he becomes its owner, an entry is created naming the owner and giving him T = 1 permission from ring 0. The process directory contains process-wide data bases, such as the procedure stack for rings 0 and 4, and the identifier table. It also contains a directory named "temp" which is used to contain the temporary segments needed by the process, such as the stacks for rings 1, 2, 3, 5, 6 and 7 and the per-ring copies of linkage segments. All segments in "temp" are destroyed when the user releases control of a process and logs-out of the system.

## Text and linkage

When the CIMPL compiler acts upon a source program both a text and a linkage segment are produced. If the branch name of the procedure segment is to be "alpha" then the text segment is given branch name "alpha$text" and the linkage segment "alpha$link". The text is pure procedure and may be shared by all users. The linkage information, however, is not pure, and each process using the procedure must have a private copy for each ring in which the procedure may execute. Thus, the linkage segment produced by the compiler is considered a master copy, and is made readable from all rings from which the associated procedure may be executed by all those who may share the procedure, but not writeable from any ring. When a process references the procedure a temporary segment is created and the master linkage segment is copied into it. Such a temporary segment is created for each ring that the procedure executes in, and is named "temp.alpha$link$ringno", where "ringno" is the specific ring.

Date: DEC 16 1968

## Identification

Memory map

## Purpose

The memory map records which blocks of memory are in use, and which are free. It is used by the memory manager of the SMS to assign blocks to and accept released blocks from the segment builder.

## Introduction

The Clics system uses a block allocation scheme in making the resource of memory space available to users. The 67,108,864 words of each memory module are divided into 262,144 blocks of 256 words. Each block may potentially contains a single page of a segment, or a single page map. All requests for memory space by the segment builder result in the assignment of a single block. Likewise, memory is returned to the free pool in the unit of a block.

In order not to permanently use a large area of memory in recording which blocks are free, and which are in use, record of this is maintained by threading together (by absolute address) all blocks not in use. The first word of each block contains the absolute address of the first word of the next on the thread. A very short segment, called the memory map, then contains the absolute address of the head of this list, and a lock which prevents simultaneous access to it by several processes. When the system is initialized all blocks not occupied by segments loaded with the system are threaded to this list.

## Memory map declaration

The memory map segment, with branch name "mem_map", is created at system initialization time and points to the head of the thread of free blocks. It is referenced by the memory manager with the following declaration:

```
declare 1 memory_map based,
          2 lock integer,
          2 free_list_head pointer;
```

The lock is set by a call to the locker before the free list is manipulated. The head of the free list is indicated with an absolute pointer. A block is always assigned from the head of the free list, and replaced there when released. (The last block on the free list contains a null pointer in its first word.)

## Identification

Memory manager

## Purpose

The memory manager of the SMS is responsible for allocating the resource of memory. Entries are provided for assigning and releasing blocks. The memory map is used to locate free blocks for assignment and record released blocks.

## Introduction

The assign and release entry points of the memory manager are called by the segment builder as it manipulates the physical structure of segments. The memory manager responds by providing, or accepting,a block of memory. Because several processes may simultaneously execute in the segment builder, they may also execute in the memory manager at the same time (on different processors). To prevent them from interfering with one another in the memory map, each is required to set a lock before referencing it. The locker is called to set this lock. (See notebook secton on the locker.) A link to word zero of "mem_map", snapped when the system is loaded, provides the pointer needed to reference the memory map segment.

## Entry points

Following is a description of the two memory manager entry points. The procedure segment is named "memory_manager".

1. assign - This entry point is defined with the following CIMPL statement:

   ```
   assign: entry (BLOCK_ADDR pointer);
   ```

   The locker is called at its lock entry point to set the memory map lock. Upon return BLOCK_ADDR is set from the memory map variable indicating the head of the free list, which is, in turn, reset from the zeroth word of the assigned block. The unlock entry point in the locker is then called to reset the memory map lock and a return made.

2. release - This entry point is defined with the following CIMPL statement:

   ```
   release: entry (block_addr pointer);
   ```

   The block designated by absolute pointer is returned to the head of the free list by the reverse of the assign procedure. The words of the returned block are set to zero (except the zeroth word, which is set with an absolute pointer to the old free list head). The memory map lock must be used.

## Identification

Segment structure

## Purpose

The segment builder constructs and destroys segments by manipulating their page maps. This section discusses the way in which page maps are referenced and the structure of the page maps of a segment may be altered by the segment builder.

## Introduction

The notebook section on address mapping in processors describes the way in which page maps enable the address mapping logic to find a given word within a segment occupying many non-contiguously located blocks of memory. Also discussed is the fact that segments may have zero, one or two levels of page maps. The segment builder constructs segments by preparing the proper structure of page maps. Depending on the initial length of the segment, this structure has zero, one or two levels.

Later in the life of a segment the segment builder may be asked to increase or decrease its length. In the simplest cases the change in length does not change the number of levels in the page map structure. Some alterations, however, do change the number of levels. Following is a description of the way page maps are referenced by the segment builder, and the technique used to alter the length of segments so that information contained in them does not need to be moved.

## Page map declaration

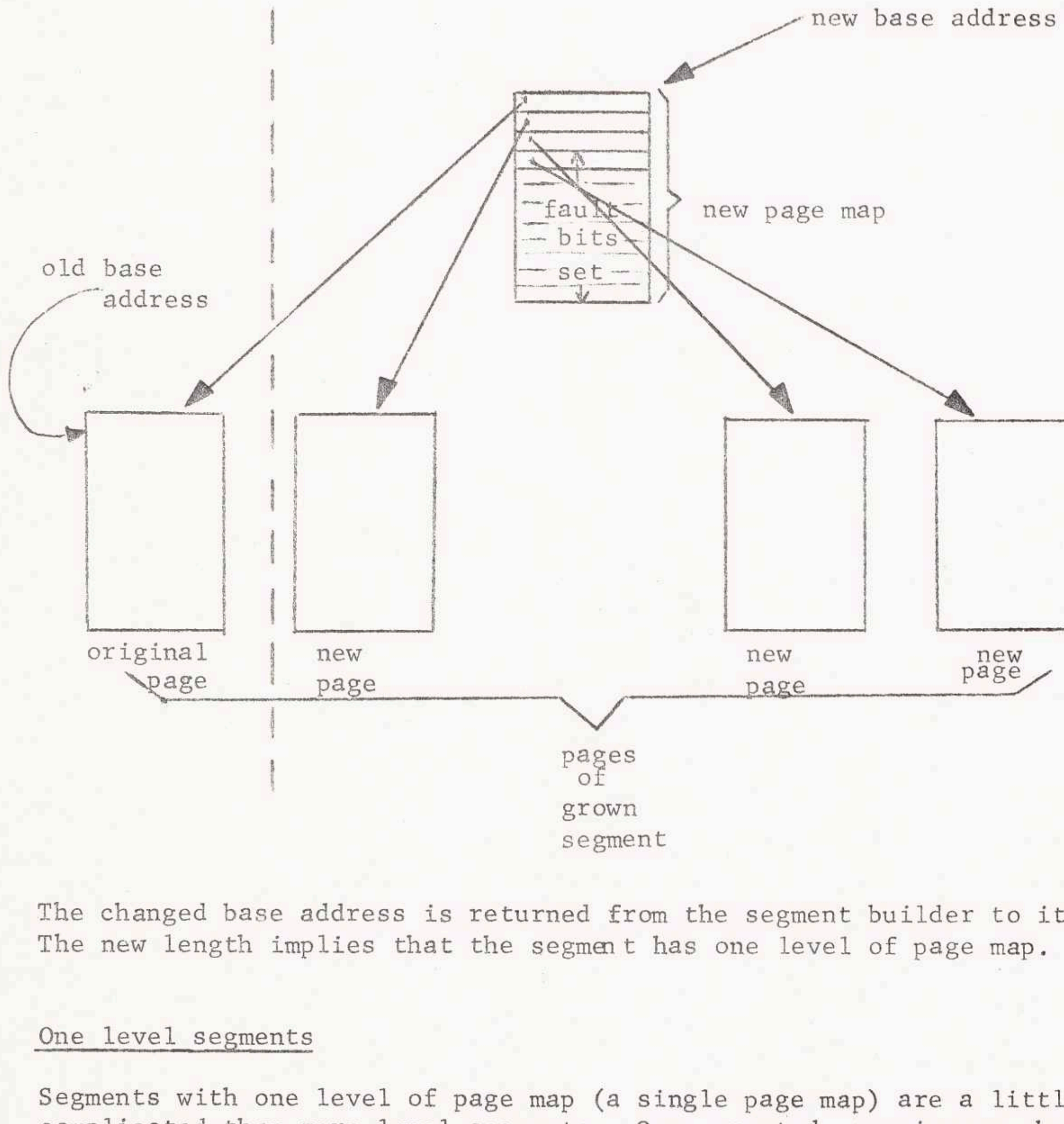
The segment builder references page maps with a based structure which is declared in CIMPL as follows:

```
     declare 1 page_map_word(256) based,
               2 empty1(11) bit,
               2 fault bit,
               2 empty2(12) bit,
               2 address(40) bit;
```

The declaration is applied with an absolute pointer which contains the address of the first word of the page map block. The bits within each word of the declaration match the processor format for page map words.

## Zero level segments

The simplest segments do not have page maps at all. Their single block is addressed directly by their base address. Of course, such segments are limited to 256 words. Alterations to the length of such segments to other values also less than 257 (including 0) do not result in a call to the segment builder at all. If the length is increased to more than 256 words, however, the segment builder is invoked to provide a page map structure. The original page of the segment is kept as the zeroth page of the grown segment and fresh blocks are assigned for the other pages and page maps. Page map words which would point to pages outside the length of the grown segment are set with fault bits. The following diagram illustrates a segment of less than 256 words that has been grown to 1000 words.



The changed base address is returned from the segment builder to its caller. The new length implies that the segment has one level of page map.

## One level segments

Segments with one level of page map (a single page map) are a little more complicated than zero level segments. Once created, growing or shrinking them between the limits of 257 and 65,536 words is accomplished by assigning or returning an appropriate number of blocks and altering page map words within the single page map. The base address of the segment, which is the address of the first word of the page map, does not change. If a one level segment is shrunk to less than 257 words, however, the base address does change. Such shrinking is accomplished by changing the base address to the address of the first word of the zeroth page, and releasing the blocks occuplied by all other pages and the page map.

Growing a one level segment to a length that requires two levels of page maps is done in much the same way that zero level segments are grown to one level. The original segment, complete with page map is incorporated into the two level structure as the page map and pages pointed to by the zeroth word of the master page map. The base address of the segment changes from that of the page map to that of the master page map. Master page map words and page map words that would point to page maps or pages that are not within the length of the grown segment are set with faults.

## Two level segments

At this point, the manner in which two level segments are grown and shrunk has become obvious. The base address will change only when the segment is shrunk to a new length that requires one or zero levels or page maps.

## Computation of levels from length

The number of levels of page maps a segment has is not explicitly contained in the segment catalog entry for a segment, and is not provided to the segment builder when it is asked to alter a segment's length. Rather, it is computed from a segment's length (which is provided). The method is very simple. Segments with length 0 through 256 have zero levels, with length 257 through 65,536 have one, and with length greater than 65,536 have two.

## Identification

Segment builder

## Purpose

When called at its three entry points the segment builder builds and destroys segments and changes their length. It calls upon the memory manager to obtain and discard blocks of memory.

## Introduction

The segment catalog manager calls upon the segment builder to physically create segments. The only specification given is the length, in words, of the new segment. The specification returned is the absolute base address of the newly created segment. The segment builder may then later be asked to alter the length of the segment. The segment to be manipulated is identified by absolute base address and current length, as segment identifiers or tree names have no meaning at the primitive level of the SMS. If the change in length alters the segment's base address, the new base address is returned. Finally, the segment builder may be asked to destroy a segment identified by base address and length.

## Entry points

Following is a description of the three entry points to the segment builder. The procedure segment is named "seg_builder".

1. build_seg - This entry point is defined with the following CIMPL statement:

```
build_seg: entry (length integer, BASE_ADDR pointer);
```

A segment of the specified length is created and its absolute base address is returned as an absolute pointer. The memory manager entry point, assign, is called to obtain the blocks required for pages and page maps. The manner in which the new segment is constructed is described in the notebook section on page maps.

2. change_seg_length - This entry point is defined with the following CIMPL statement:

```
change_seg_length: entry (base_addr pointer, old_length integer,
new_length integer, NEW_BASE_ADDR pointer);
```

The length of the segment identified by absolute pointer to its base and its length is changed to the new length specified. The corresponding alterations to the segment's structure are described in the notebook section on segment structure. The new base address of the segment is returned as an absolute pointer. (This may be the same as the old base address.) The assign or release entry point in the memory manager is called to obtain new blocks for pages and page maps, or to release unneeded blocks. If the new length specified is zero, it is processed as though it were one through 256 - a segment of a single page and zero levels of page maps is retained.

3. destroy_seg - This entry is defined with the following CIMPL statement:

```
destroy_seg: entry (base_addr pointer, length integer);
```

The segment identified is destroyed. Blocks occupied by pages and page maps are returned by calling the release entry point in the memory manager.

## Identification

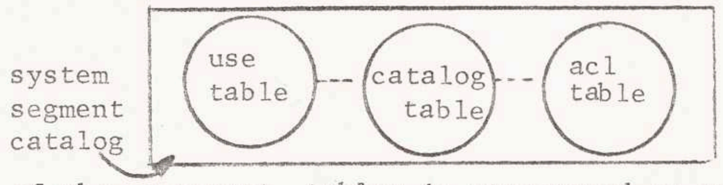
System segment catalog

## Purpose

The system segment catalog is the SMS data base that contains an entry for each segment in the Clics system. Segments in the catalog are located by unique system-generated segment identifier. An entry contains the length and absolute base address of the corresponding segment, the access control list of the segment, and the list of processes using the segment i.e., of processes that have the segment in their address space. The system segment catalog is referenced by the segment catalog manager of the SMS.

## Introduction

The system segment catalog is organized as three separate segments: the catalog table, the use table, and the access control list table. It may be diagrammatically represented as:



Each of the component tables is structured as an array of equal length blocks (not to be confused with the 256 word memory blocks). The blocks may be accessed by index once a pointer to the zeroth word of the segment is available. A block in the catalog table may become the head of a system segment catalog entry, and contain the identifier, length, and absolute base address of the described segment. It can also be linked by use table block index to the head of the chain of use table blocks describing processes using the segment, and by acl table block index to the head of the chain of acl table blocks containing access control list entries for the segment.

The segment identifier in the head of an entry is the calendar clock reading taken at the time the entry is created. Since this microsecond clock can only be read once per microsecond, this reading is guaranteed to be unique

All three segments are created during system initialization, and originally contain record of all segments loaded with the system.

The branch names of the three segments are "cat_tbl", "use_tbl" and "acl_tbl".

## Catalog table declaration

The catalog table segment is referenced by the segment catalog manager with the following CIMPL declaration. The pointer needed for each reference is obtained from a link snapped during system initialization.

```
     declare 1 cat_tbl based,
               2 num_blocks integer,
               2 block(*),
                 3 seg_id integer,
                 3 base_addr integer,
                 3 length integer,
                 3 use_list integer,
                 3 acl integer,
                 3 ent_lock integer;
```

The list below describes the items in the declaration.

num_blocks - number of blocks contained within the length of the catalog segment.

block(*) - array of catalog blocks. Each block is either empty or contains a system segment catalog entry head.

seg_id - If block is not in use then contains zeros. If block is catalog entry head, then contains identifier of corresponding segment. The format of identifiers is described above.

base_addr - integer that is the absolute base address of the corresponding segment.

length - length, in words, of corresponding segment.

user_list - index of block in use table that is the first in the chain of use table blocks recording processes using the segment corresponding to the catalog entry. (If list is empty then contains -1.)

acl - index of block in acl table that is the first in the chain of acl table blocks containing the corresponding access control list. (If it is empty then contains -1.)

ent_lock - entry lock which must be set through the locker's entry point lock before entry can be read or written.

## Use table declaration

The use table segment is referenced by the segment catalog manager with the following CIMPL declaration. The pointer needed for a reference is obtained from a link snapped during system initialization.

```
declare 1 use_tbl based,
          2 lock integer
          2 free_list integer,
          2 num_blocks integer,
          2 block(*),
            3 dbr_image integer,
            3 segno integer,
            3 link integer;
```

The list below describes the items in the declaration:

lock - manipulation lock. Must be set through the locker's entry point lock before free list is manipulated.

free_list - index of first block in chain of empty use table blocks.

num_blocks - number of blocks contained within the length of the use table segment.

block(*) - array of use table blocks. Each either contains record of use of a segment by a process, or is on the list of free ( empty) blocks.

dbr_image - contains the image of the descriptor base register of a process using a segment.

segno - contains the segment number assigned the segment within that process.

link - has two purposes. If block is on the free list then contains the index of the next block on the free list. If block is in use then contains the index of the next block in the list for the same segments.

The free list is constructed at system initialization to contain all unused blocks. Block assignment and release occurs at the top of the free list under protection of the lock.

## acl table declaration

The acl table segment is referenced by the segment catalog manager with the following CIMPL declaration. Again, the pointer needed for a reference is obtained from a link snapped during system initialization.

```
declare 1 acl_tbl based,
          2 lock integer,
          2 free_list integer,
          2 num_blocks integer,
          2 block(*),
            3 user_name(32) character,
            3 ami(11) bit,
            3 link integer;
```

The table header items are used as previously described. Each block is either on the list of unused blocks, or contains a single access control list entry for a segment. The items in a block are:

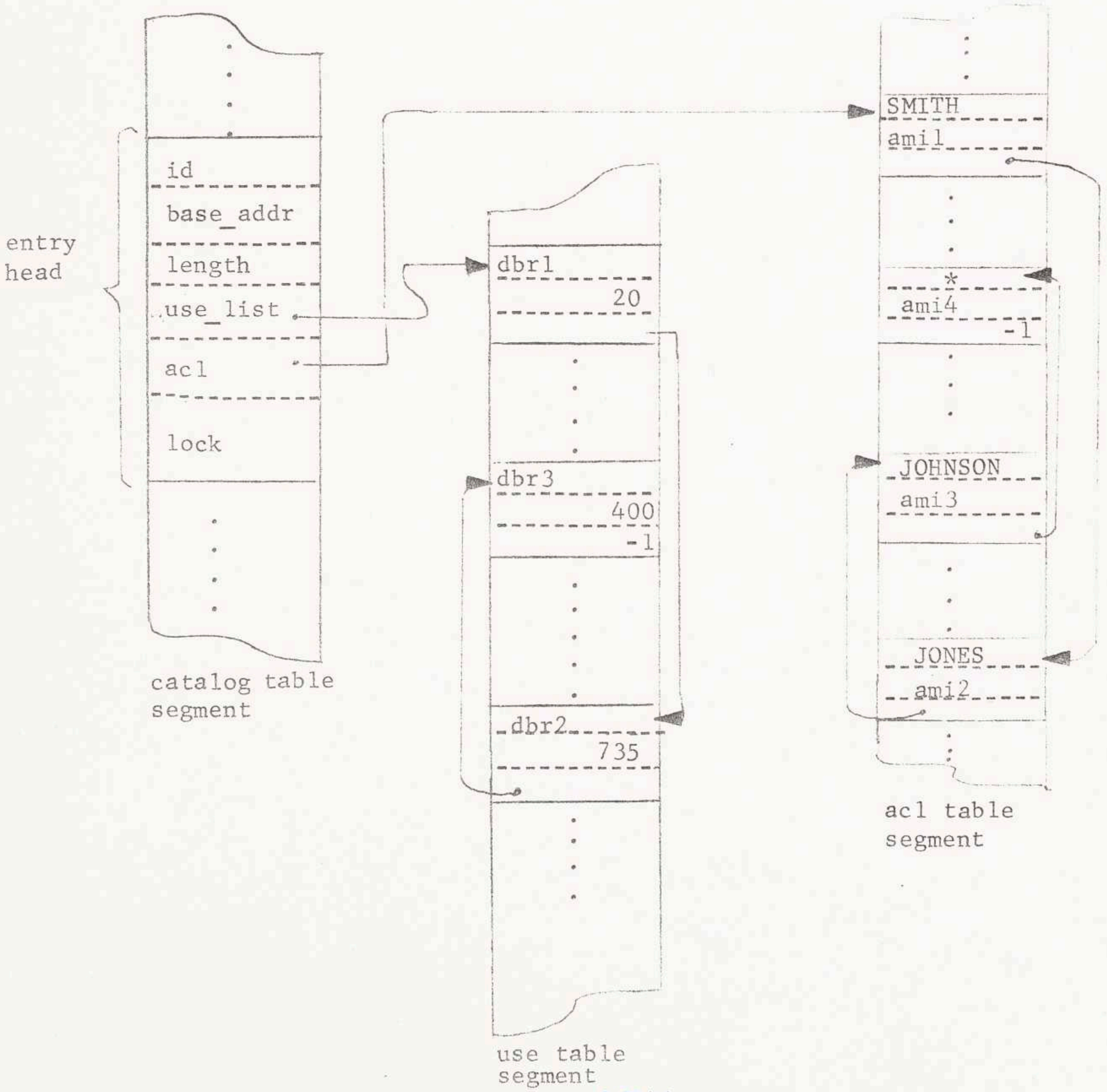
user_name - a 32 character system user identifier.

ami - the access mode indicator applying to that user when he causes his process to use this segment.

link - two uses. If block is empty then contains the index of next block on free list. If block contains an access control list entry for a segment then contains index of next entry on that segments' access control list.

## Typical system segment catalog entry

The diagram below illustrates a typical entry in the system segment catalog. Note the convention of using the index minus one to indicate the end of an index linked chain.



The segment discribed by the entry has three processes using it. The first knows it as segment number 20, the second as 735, and the third as 400. Three users are given specific permission to cause their processes to include the segment in their address spaces, and an individual access mode indicator is specified for each. All other users are given access under a further access mode indicator.

## Identification

Segment catalog manager

## Purpose

The segment catalog manager is the only procedure module in the segment control portion of the SMS. Its purpose, as suggested by its name, is to make all references to and to perform all manipulations of the system segment catalog. It is called by both the directory manipulator and the process address space manager. It calls upon the segment builder of memory control to perform physical manipulations on segments.

## Introduction

The segment catalog manager maintains in the system segment catalog record of all segments in the Clics system. The items included in the entry for each are described in the notebook section on that data base.

The segment catalog manager accepts calls from the directory manipulator requesting creation of a new segment and corresponding catalog entry, alteration of a segment and its catalog entry, and deletion of a segment and its catalog entry. It will also list the contents of a catalog entry. The caller specifies the segment involved by segment identifier. Any implied alterations to the physical structure of a segment, including creating a new segment, are carried out by calling the appropriate entry points in the segment builder.

A second group of calls is accepted from the process address space manager. Information from which to build a segment descriptor word (SDW) for a segment specified by segment identifier may be requested. Later, when the segment is removed from a process' address space, another call informs the segment catalog manager that the segment's use record should be updated.

The segment catalog manager calls upon the process address space manager to set SDW fault bits "on" when it determines that the information in a process' SDW for a segment has become invalid.

## Block allocation

As described in another notebook section, the three tables that comprise the system segment catalog are organized as arrays of blocks. Unused blocks in the access control list table and the use table are threaded together into a list of free blocks. When it becomes necessary to assign a block in one of these two tables the block at the head of the proper free list is used. Likewise, when a block is released it is replaced at the head. Released blocks are set to zero except for the index that is the free list thread. The lock in the appropriate table header must be set through the lock entry point in the locker before free list manipulations can occurs and reset after manipulation with the unlock entry point.

A free list is not used in the catalog table. Here empty blocks can be recognized because they contain all zero identifiers. An empty block in which to build a system segment catalog entry head for a new segment is located and reserved by making a linear search of the catalog table block array. When a block is encountered whose identifier is all zero it is locked by calling the lock entry point in the locker. Then the identifier is again checked for all zeros, and if such is found the locked block is used for the new entry. If all zeros is not found on the second check the block's lock is reset using the unlock entry point in the locker and the search containues.

When a system segment catalog entry head is destroyed the block occupied is returned to the free pool by resetting it to zero. The block's lock is the last word reset, and is reset by calling the unlock entry point in the locker.

## Catalog entry location

All calls to the segment catalog manager besides the one which creates a new segment include as an argument a segment identifier. The first task of each of these cases is to locate and lock the corresponding entry head in a block of the catalog table. The procedure followed is to make a linear search of the table beginning with block(0). At each increment of the search the procedure is to match the given segment identifier with that in the block. If they do not match the index is advanced and the next increment in the search begun. If they do match then the block's lock is set by calling the lock entry point in the locker, and the identifers are compared again. If still equal the desired entry head has been found and locked. If different the second time then the desired entry head does not exist, and the search may be stopped. (Of course, if the end of the table is reached without making a first match then the entry does not exist either.)

If the required entry is located and locked, then its lock is reset after the required manipulations are completed by calling the unlock entry point in the locker.

## Entry points

Following is a description of the seven entry points to the segment catalog manager. The procedure segment has the name branch name "s_cat_mgr"

1. create_seg - This entry point is defined with the following CIMPL statement:

```
        create_seg: entry (length integer, SEG_ID integer);
```

The first argument specifies the length for a segment to be created. The build_seg entry in the segment builder is called to build a new segment of the specified length, and then return its base address. An empty catalog block is found and locked for the new system segment catalog entry head, and set with the length, the base address, and a segment identifier constructed as described in the notebook section on the system segment catalog.

Finally, the use list head index and the acl list head index in the entry head are set to minus one (to indicate that they are empty), the entry lock is reset, and the constructed segment identifier is returned.

2. change_seg_length - This entry point is defined with the following CIMPL statement:

```
change_seg_length: entry (seg_id integer, new_length integer);
```

The catalog block containing the system segment catalog entry head matching the given identifier is located and locked. The current length of the segment is compared with the given new length. If the change will not alter the the segments' structure (i.e. cause at least one memory block to be added or removed) then the length in the entry is replaced with the new length, the entry unlocked, and a return made. If the change will alter the segment's structure but not change the number of levels of page maps then the change_seg_length entry point in the segment builder is called to restructure the segment, the new length stored in the entry, the entry unlocked, and a return made. If the change will increase or decrease the number of levels of page maps then current users must first be informed. The set_fault entry point in the process address space manager is called for each entry on the use list attached to the entry head. The arguments given are exactly the contents of each use list entry. The entire use list is then destroyed by releasing all the use table blocks occuplied and resetting the index to its beginning in the entry head to minus one. The segment builder is then called to make the necessary changes to the segment's physical structure, the new length and returned new base address are set in the entry head, the entry lock reset, and a return made to the caller.

3. delete_seg - This entry point is defined with the following CIMPL statement:

```
delete_seg: entry (seg_id integer);
```

The catalog block containing the system segment catalog entry head for the given identifier is located and locked. The entire entry and the corresponding segment are then destroyed by calling the set_fault entry point in the process address space manager for each entry on the attached use list, destroying the use and access control lists by returning the occupied use table and acl table blocks, calling the destroy_seg entry point in the segment builder to destroy the segment and releasing the catalog block containing the entry head.

4. change_acl - This entry point is defined by the following CIMPL statement:

```
change_acl: entry (seg_id integer, ins_del_sw bit, user_name(32)
  character, ami(11) bit);
```

The catalog block containing the system segment catalog entry head for the given identifier is located and locked. The insert/delete switch given as the second argument is inspected. If it is "off", indicating insertion, then a block in the acl table is allocated and linked to the end of the access control list of the segment. The third and fourth arguments are set in this block as a user name and the access mode indicator. If the switch is "on", indicating deletion, then the access control list entry for the user named by the third argument is removed from the segment's access control list by relinking the list around it and releasing the occupied block. Finally, the entry head lock is reset and a return made.

5. list_entry - This entry point is defined by the following CIMPL statement:

```
          list_entry: entry (seg_id integer, target pointer);
```

The block containing the system segment catalog entry head for the identified segment is located and locked. The pointer provided as the second argument is assumed to the point to a 2503 word area, and is applied to the following CIMPL based structure:

```
     declare 1 catalog_entry based,
                2 length integer,
                2 num_users integer,
                2 num_acl_entries integer,
                2 acl (500),
                  3 user_name (32) character,
                  3 ami (11) bit;
```

The length of the identified segment is copied into "target → catalog_entry.length", the number of entries on the use list is counted and stored in "target → catalog_entry.num_users", and the contents of each entry on the segment's access control list is copied into an element of the "target → catalog_entry.acl" array. The length of the array is also copied into the proper structure element. If there are more than 500 access control list entries then only the first 500 are returned. The entry head is then unlocked and return made.

6. initiate: entry (seg_id integer, segno integer, val_lvl integer,
AMI(11) bit, LEVEL integer, ADDRESS integer);

The first argument specifies a segment that a process wishes to include in its address space, the second the segment number that will be assigned it, and the third the ring occupied by the procedure wishing to reference it. Information from which to build a segment descriptor word for the segment is returned if the following three conditions are met:

a) system segment catalog entry corresponding to the identifier can be found.

b) owner of process is on the segment's access control list.

c) ring occupied by procedure wishing to reference segment is less than or equal to at least one of R1, R2 and R3 from appropriate access mode indicator.

If some condition is not met then an ADDRESS of -1 is returned instead.

The procedure is to locate and lock the corresponding system segment catalog entry head, and determine the name of the owner of the requesting process from the process table entry of the process in which execution is occurring. The access control list is then searched for a matching entry, and the contained access mode indicator inspected to verify the segment will be accessible from the ring given as the third argument. Next a block in the use table is assigned and linked to the head of the segment's use list. Saved in this block are the contents of the processor's descriptor base register, and the segment number to be assigned the segment. Finally, the segment's absolute base address, level number, and the access mode indicator are returned after unlocking the catalog entry.

7. terminate - This entry point is defined with the following CIMPL statement:

```
        terminate:  entry (seg_id integer);
```

The system segment catalog entry head for the identified segment is located and locked. If none is found a return is made to the caller. The use list attached is then searched for an entry containing a descriptor base register image matching the current processor descriptor base register. If none is found then the entry head is unlocked and a return made. If an entry is found that matches then it is deleted from the list by relinking around it and releasing the block occupied. Then the entry head is unlocked and a return made,.

## Identification

Descriptor segment

## Purpose

The descriptor segment of a process defines its address space by containing a segment descriptor word (SDW) for each segment contained therein. The SDW is used by the address mapping logic of the active modules in the hardware to transform addresses given as (segment number|word number) into absolute form. The SDW corresponding to a given segment number is easily located, as the segment number is the required offset in the descriptor segment. Each time that a new segment is added to an address space a new SDW must be created.

## Introduction

It is the responsibility of the process address space manager of the SMS to alter the descriptor segment of a process. Two types of alterations are performed. The normal case is to alter the descriptor segment of the process in which the process address space manager is executing. In this case the descriptor segment is referenced through a pointer to it obtained from a link which was snapped when the process was initialized. (The descriptor segment of a process has the branch name "desc_seg" and appears in the process' process directory.) The less frequent case is to alter a SDW in the descriptor segment of another process. In this case the SDW is referenced through an absolute pointer constructed from an image of the process' descriptor base register and the segment number.

The descriptor segment of a process is created with the process and initialized to contain in words 0 through n-1 the SDW's for the n segment supplied with the process. The remaining words of the descriptor segment are initialized with the fault bit set "on".

## Declarations and discussion

Because the process address space manager has the ability to reference the descriptor segment of an arbitrary process, as well as the descriptor segment of the process in which it is executing, it is possible that the descriptor segment of process A can be simultaneously referenced by process A executing in the process address space manager and process B, also executing there. Confusion could result from this if process A had gotten there by calling either the process_sdw_fault entry point of the get_ami entry point. (Process B had to have gotten there by calling the set_fault entry point.)

The following declarations are used by the process address space manager to prevent the possible confusion in those two cases. To keep the total number of declarations used as small as possible and to make the entry points of the process address space manager consistent they are used in all cases, although other more obvious means of referencing SDW's would work in most cases.

The descriptor segment of the process in which the process address space manager is executing is referenced through the following based structure:

```
     declare 1 desc_seg(*) based,
               2 sdw(64) bit;
```

Only two kinds of references are allowed. The 64 bits of a single SDW can be read at once, or can be written at once (the latter using either a normal store instruction or the special store if operand zero instruction). The following automatic structure is used as the target and source of such reads and writes"

```
     declare 1 sdw,
               2 ami(11) bit,
               2 fault bit,
               2 empty(10) bit,
               2 level(2) bit,
               2 address(40) bit;
```

The structure includes 64 bits, and matches the fields of an SDW.

The process address space manager references an SDW in the descriptor segment of another process through the following based array:

```
     declare SDW2(64) based;
```

The needed pointer is absolute and produced from a descriptor base register image for the other process and the segment number corresponding to the SDW. It is used only to write the following pattern of 64 bits into an SDW:

```
    0000000000010 ... 0
```

which has the effect of zeroing the SDW and setting the fault bit "on".

## Identification

Identifier table

## Purpose

The identifier table is a per-process data base whose entries are in one-to-one correspondence with the words of the process' descriptor segment. In each entry is recorded the system segment catalog identifier of the segment described by the corresponding segment descriptor word. The three word header of the identifier table controls the assignment of segment numbers, and specifies which segments in a process' address space were placed there by the system, and which were included directly or indirectly by the action of user procedures. The identifier table is referenced by the process address space manager.

## Discussion

Following is the declaration of the identifier table that appears in the process address space manager. The segment containing the table itself is created when the process is created at system initialization time, and initialized to describe the segments supplied with the process. The pointer required to apply the declaration is obtained from a link snapped when the process is initialized. The identifier table of a process has the branch name "id_tbl" and appears in the process' process directory.

```
declare 1 id_tbl based,
          2 next_segno integer,
          2 capacity integer,
          2 first_user_seg integer,
          2 id(*) integer;
```

The following list describes the items in the declaration:

next_segno - This variable contains the next segment number to be assigned by the process. At each assignment it is incremented by one.

capacity - This variable contains the current length of the process' descriptor segment.

first_user_seg - This variable marks the beginning of the segments that have been included in a process' address space during and after it has been assigned to a user. When the user logs-out of the system, releasing control of the process, all these segments are removed from the address space.

id(*) - This array of integers is in one-to-one correspondence with the words of the descriptor segment. Each contains the identifier of the segment described by a segment descriptor word. An identifier of zero signifies that the corresponding descriptor segment word has not been assigned.

## Identification

Process address space manager

## Purpose

The process address space manager, the only procedure segment in the address space control portion of the SMS, manipulates the address space of a process. Always linked to the descriptor segment and identifier table of the process in which it is executing, it makes entries there that allow other procedures in the process to reference segments from the system segment catalog. A restricted mechanism for removing the segments from an address space is also provided. The process address space manager is additionally responsible for processing segment descriptor faults. It is called by the directory manipulator and certain modules in the processor management subsystem; and calls the segment catalog manager. The latter may also call back,using the process address space manager to set segment descriptor word (SDW) fault bits "on" in arbitrary process' descriptor segments.

## Introduction

A descriptor segment defines the address space of each process, containing an SDW for each included segment. The companion identifier table entry contains the identifier of the segment described. When a call is received from the directory manipulator to get a (segment number|word number) pointer to the zeroth word of a segment, one of two situations can occur. If the identified segment is already in the process' address space then its identifier will be found by linear search in the identifier table. The required pointer can be directly constructed and returned. If the segment is not already in the address space, it must first be included there. The segment catalog manager provides the necessary SDW information from which to build the SDW required.

Within the header of the identifier table is a variable indicating the next segment number to be assigned. With each inclusion of a segment into the address space this is incremented. The only way that segments may be directly removed from a process' address space is through an entry point that can perform the single function of removing all segments that have been included in the address space since the process was assigned to its current owner. When such housekeeping does occur the segment number assignment variable is reset. This entry can only be called from within ring zero.

In processing a segment descriptor fault the process address space manager determines whether the fault was caused by a previously valid SDW, and, if so, attempts to re-include the corresponding segment into the address space under the same segment number.

When performing the function of setting the SDW fault bit "on" in an arbitrary process' descriptor segment, the process address space manager simulates the address mapping logic of the processor to convert the supplied descriptor base register contents and segment number into the absolute address of the SDW to be operated upon. (Allowing the hardware to compute the absolute address would involve loading the descriptor base register image into the processor's descriptor base register, and thus would mean switching processes.)

Entry points

Following is a description of the five entry points in the process addresd space manager. The procedure segment is named "pas_mgr".

1. get_ptr - This entry point is defined with the following CIMPL statement:

```
        get_ptr: entry (seg_id integer, val_lvl integer, PTR pointer);
```

The first argument specifies the segment to which a pointer is desired. The identifier table is searched linearly for a matching identifier. If found, the entry index is the required segment number. The get_ami entry point in this same module is called to obtain the access mode indicator for the segment, and it is verified that at least one of R1, R2 and R3 contained therein is greater than or equal to the ring of the procedure that is to reference the segment, as given by the second argument. If not, then a null pointer is returned. If so then a pointer to word zero of the segment is constructed and set as the third argument and a return is made to the caller. If no matching identifier is found in the identifier table then a new segment number is tentatively assigned as that specified in the identifier table header as the next segment number to be used. The initiate entry in the segment catalog manager is called with the identifier and tentative segment number as arguments. If the returned base address is not minus one then the return information is used to create a new SDW in the temporary structure that matches a SDW's format. The SDW is then stored into the proper descriptor segment word. The corresponding identifier table entry is set with the segment's identifier, and the segment number assignment variable is incremented. Then a return is made to the caller with a pointer constructed from the segment number. If the returned base address is instead minus one then a return is made to the process address space manager's caller with a null pointer.

2. set_fault - This entry point is defined with the following CIMPL statement:

```
        set_fault: entry (dbr_image integer, segno integer);
```

The first argument is interpreted as the absolute base address of a process descriptor segment. The action of the processor address mapping logic is simulated to produce the absolute address of the SDW for the segment whose number within that process is given as the second argument. The fault bit in that SDW is then set "on" by storing a word of zeros, except for bit 11 which is one, into the location indicated by the absolute address.

3. remove_user_segs - This entry point is defined with the following CIMPL statement:

```
        remove_user_segs: entry;
```

The variable in the header of the identifier table that contains the segment number beyond which all segments have been included in the address space since the current owner gained control of the process is inspected. For each segment number between this and the number indicated by the variable specifying the next segment number to be assigned, the following action is taken.

a) The corresponding segment identifier is determined by referencing the matching identifier table entry.

b) The terminate entry point in the segment catalog manager is called with the identifier as an argument to update the corresponding segment's use record.

c) The segment's SDW is set to zero with the fault bit "on" (by storing the bit pattern 0000000000010...0 into the SDW).

d) The identifier table entry is set to zero.

When this is completed for all segments in the specified group the variable specifying the next segment number to be assigned is reset to the smallest segment number that was terminated.

4. process_sdw_fault - This entry point is defined with the following CIMPL statement:

```
        process_sdw_fault: entry (segno integer, RESULT bit);
```

The first argument identifies the word of the descriptor segment that caused the segment descriptor fault. If the corresponding identifier is zero then a return is made with a RESULT of "0"b. If an identifier is found to correspond to the faulting word then an attempt is made to reinclude the particular segment in the process' address space.

This involves the following steps:

a) Set the faulting SDW to all zero.

b) Call the initiate entry point in the segment catalog manager to get new SDW information. If an address of minus one is returned then set the SDW fault bit "on" again and return with RESULT set to "0"b.

c) Format the new SDW in a temporary variable and store it into the descriptor segment with an "stoz" machine instruction. If successful, return with the RESULT set to "1"b. If unsuccessful go back to step a.

This rather elaborate procedure is necessary to guarantee that if the segment in question is deleted by its owner between the call to initiate and the construction of the new SDW the fault bit will be properly set "on".

5. get_ami - This entry point is defined with the following CIMPL statement:

```
        get_ami:  entry (segptr pointer, AMI(11) bit, NO_AMI_SW bit);
```

The segment number in the given pointer is determined and a check is made to verify that it falls within the current length of the descriptor segment. Then the corresponding SDW is read into the temporary SDW structure. The SDW fault bit is checked, and if "off" the access mode indicator is returned, with the third argument set to "0"b. If "on" then the process_sdw_fault entry point to this same module is called to attempt resetting it. A returned RESULT of "0"b causes a return to the caller with the third argument set to "1"b. A result of "1" b causes this entire procedure to be repeated.

## Identifier

Directory segments

## Purpose

Directory segments are the building blocks of the SMS hierarchy. Created, manipulated, and destroyed at the direction of properly authorized system users, they may be directly referenced only by the directory manipulator of the SMS. This notebook section desctibes the structure of a directory segment.

## Introduction

A directory contains branches for both directory and non-directory segments. The number of each varies over the life of a directory as branches are added and deleted. Non-directory branches are of a fixed size, but directory branches may contain arbitrarily long directory control lists that can be altered during the life of the branch. Because of this need for a directory to gracefully contain a variable number of variable length items, space is allocated and released on a block basis. Each directory block may be in one of three states: unused, containing a directory or non-directory branch header, or containing a directory control list entry.

Directories are created and referenced by the directory manipulator. This module is presented with a pointer to the zeroth word of the directory to be referenced, and is unaware of the hierarchical structure relating all segments of the system.

## Directory declaration

The following CIMPL declaration is used by the directory manipulator to reference a directory when supplied with a pointer to its zeroth word.

```
     declar 1 dir based,
              2 lock integer,
              2 capacity integer,
              2 free_list integer,
              2 branch_list integer,
              2 dir_id integer,
              2 block(*),
                3 link integer,
                3 name(32) character,
                3 seg_id integer,
                3 dir_sw integer,
```

When the directory manipulator creates a new directory the five items in the header are initialized as follows:

| item | value |
|---|---|
| lock | zero |
| capacity | number of blocks within directory segment's length |
| free_list | zero |
| branch_list | minus one |
| dir_id | segment identifier of this directory segment |

The first, the directory lock, will be described later in this section. The second indicates how many blocks of 7 words are contained within the length of the directory segment. The third specifies that block(0) is the first on the list of unused directory blocks. This list is linked by block index in the first word of each block. Initially, all directory blocks are on it. The fourth will become the index of the first in a chain of blocks containing branch headers. The fifth is the system segment catalog segment identifier of this directory segment.

Assignment of blocks is done from the top of the free list. The index in "p → dir.free_list" is that of the block assigned. The new free list head is taken from the first word of the assigned block. Release blocks are attached to the head of the free list by the reverse process. If the free list becomes empty (indicated by an index of -1 in its head indicator) it signifies that the directory is completely full. Attempted block assignment will cause an error to be returned to the caller.

As blocks are assigned to contain branch headers, they are linked together through their first word onto the branch list. This list, whose head is indicated by index in the directory header, is searched whenever a branch with a given name is to be located. The second item in blocks containing a branch header is the 32 character branch name. This is followed by the system segment catalog identifier of the corresponding segment. The final item is a switch specifying whether the branch describes a directory or a non-directory segment. If a non-directory, it contains minus two. If it contains a value other than minus two then the branch describes a directory segment, and the value contained is the index of the first block containing a directory control list entry for the branch. (If minus one that list is empty.)

## Directory control list entries

If a directory block is to contain a directory control list entry the following structure is applied to the specific block.

```
declare 1 dcl_entry based,
          2 link integer,
          2 user_name(32) character,
          2 dai(4) bit;
```

The entries on the directory control list of a single directory branch are linked together into a chain by index in their first word. Each entry contains a user name and the corresponding directory access indicator. The use of the directory control list is described in the notebook section on the sharing and protection of segments.

## Directory lock

Each directory segment contains a lock as its first word. This lock is used to prevent interprocess interference in the directory. Each time a directory is to be accessed for any reason by the SMS the lock must be set by a call to the lock entry point in the locker. Because only one process at a time may have a lock set, only one can be executing in the directory manipulator referencing a particular directory. The lock is reset when manipulation of the directory is completed, allowing another process access to it. (A directory is locked and unlocked at the hierarchy control level of the SMS.)

## Identification

Directory manipulator

## Purpose

The directory manipulator, the only procedure module in the directory control portion of the SMS, makes all direct references to directory segments. When called by either the hierarchy access validator or the segment locator it is given a (segment number|word number) pointer to word zero of the directory segment it is to reference. It is unaware of the structure of the hierarchy beyond that which is implicit within a single directory and its branches. The directory manipulator calls upon both the segment catalog manager and the process address space manager.

## Introduction

The directory manipulator receives calls from two sources: the hierarchy access validator and the segment locator. Those from the former cause the indicated directory segment to be manipulated, possibly producing the addition of a branch or a change to an existing one. The segment catalog manager is generally called in turn to perform some alteration to the corresponding system segment catalog entry. Calls to the directory manipulator from the latter source, the segment locator, are to obtain a (segment number|word number) pointer to a segment whose branch is in the indicated directory. Its function in these cases is merely to convert the given branch name to the corresponding segment catalog identifier, and pass the call on to the process address space manager.

## Entry points

Following is a description of the eight entry points to the directory manipulator. The first two are called by the segment locator, while the remaining six are called by the hierarchy access validator. The procedure segment is named "dir_manip".

1. get_dir_ptr - This entry point is defined with the following CIMPL statement:

```
get_dir_ptr: entry (dirptr pointer, branch_name(32) character, DAI(4)
     bit, NO_DAI_SW bit, PTR pointer);
```

The first argument points to the zeroth word of a locked directory segment. The branch list of that directory is searched for a directory segment with the branch name given by the second argument. If such is not found a <u>null</u> PTR is returned. If such a branch is found then the get_ptr entry point in the process address space manager is called with the contained identifier and a ring number of zero as arguments. The returned pointer to the directory segment will, in turn, be returned as the desired PTR. Before that return is made, however, the directory control list in the branch is searched for an entry matching the owner or the requesting process. If such an entry is located then the contained directory access indicator is returned as the third argument, with the fourth argument set "off". Otherwise a return is made with the fourth argument set "on", indicating no matching directory control list entry could be found.

2. get_nondir_ptr - This entry point is defined with the following CIMPL statement:

```
        get_nondir_ptr: entry (dirptr pointer, branch-name(32) character,
        val_lvl integer, PTR pointer);
```

This entry is very similar to that described above. In this case, however, a pointer to a non-directory segment is desired. The procedure followed is similar. A branch with the given name for a non-directory segment is searched. The pointer returned from the process address space manager (which is called with the given validation level as the second argument) is, in turn, returned to the caller. If no pointer is returned, indicating that the requesting user was not on the segment's access control list in the catalog entry, or that the validation level was to high, a null PTR is returned instead.

3. create_seg - This entry point is defined with the following CIMPL statement:

```
        create_seg: entry (dirptr pointer, branch_name(32)
                character, length integer, CODE integer);
```

The arguments specify a segment to be created. The first is a pointer to the zeroth word of the directory which is to contain the new segment's branch. The contained branch list is searched for an already existing branch with the same name as that specified by the second argument for the new segment. If such exists, a return is made with an error CODE. Otherwise a directory block is assigned to contain the branch header for the new segment. (If the directory is full an error code is instead returned.) The new branch header is linked to the beginning of the directory's branch list.

At this point it is determined whether the new segment is to be a directory or a non-directory. The length given as the third argument is the clue. If minus one, then directory is the case. If greater than or equal to zero, then non-directory is the case, and the value is the initial length for the new non-directory segment.

To create a directory segment the directory manipulator calls the create_seg entry point in the segment catalog manager, specifying the standard length for new directory segment and then calls the change_acl entry to create the standard directory segment access control list with a single entry allowing all users read and write permission from ring zero. The new segments' identifier is placed in the new branch header along with the given branch name. The directory switch in the branch header is set to minus one, indicating that the directory control list is empty.(This situation can be remedied by calling the change_ctl_list entry point in the directory manipulator later.) This completes the creation of a directory segment except for the final task of initializing the new directory. To do this a pointer to it is obtained by calling the get_ptr entry point in the process address space manager with the directory's identifier as an argument. The returned pointer bases the standard directory declaration which is used to set the directory's header variables and link all blocks to the free list.

Creation of a non-directory segment is somewhat simpler. The create_seg entry point in the segment catalog manager is called to create the new non-directory. The argument for this call is the third argument in the argument list of the directory manipulator entry being described, which specifies the length for the new segment. The returned identifier and the given branch name are placed in the new branch header, and the contained directory switch is set to minus two. This completes creation of the new non-directory segment.

4. change_seg_length - This entry point is defined with the following CIMPL statement:

```
change_seg_length: entry (dirptr pointer, branch_name(32) character,
   new_length integer, CODE);
```

The length of the segment indicated by branch name within the specified directory is to be changed if it is a non-directory. (The SMS does not allow users to specify changes in the length of directory segments.) The branch list of the directory pointed to by the first argument is searched for the branch with the specified branch name. If such is not found, or the found branch describes a directory segment, an error CODE is returned. Otherwise, the call is passed on to the change_seg_length entry point in the segment catalog manager, with the given new length and the identifier from the branch as arguments. The return is made immediately following the return from the segment catalog manager.

5. delete_seg - This entry point is defined with the following CIMPL statement:

```
delete_seg: entry (dirptr  pointer, branch_name(32) character,
  CODE integer);
```

The specified branch in the directory pointed to by the first argument is to be deleted, and the corresponding segment destroyed. The branch list in the directory is searched for the matching branch. If such is not found an error CODE is returned. Otherwise, if the found branch describes a non-directory segment the branch is destroyed by returning the occupied block to the directory's free list and the segment is destroyed by calling the delete_seg entry point in the segment catalog manager with the identifier from the branch as an argument.

If the found branch describes a directory segment, however, it must be verified that the directory contains no branches before it can be deleted. The directory's identifier is passed to the get_ptr entry point in the process address space manager and the returned pointer used to base the standard directory declaration. If the branch list in the directory is indeed empty then the directory segment can be destroyed as above. (Directory control list blocks must also be released.)

6. change_ctl_list - This entry point is defined by the following CIMPL statement:

```
        change_ctl_list: entry (dirptr pointer, branch_name(32)
                   character, dir_sw bit, ins_del_sw bit, user_name(32)
                   character, indicator(*) bit , CODE integer);
```

The access or directory control list of the specified non-directory or directory segment within the directory pointed to by the first argument is to be changed. The branch list of the directory is searched for the branch named by the second argument. If it cannot be found, or does not match the type specified by the directory switch that is the third argument ("off" means non-directory, "on" directory), a return with an error CODE is made. Otherwise, changes to directory and access control list are treated separately.

If the branch and input directory switches specify a non-directory segment then the change to the access control list is caused by passing on the call to the change_acl entry in the segment catalog manager. The arguments used are the identifier from the branch and the fourth through sixth input arguments.

If the branch and input directory switch specify a directory segment then the change to the directory control list is performed directly. The insert/delete switch that is the fifth argument is inspected. If "off", then the sixth and seventh arguments are a new directory control list entry to be added to the end of the branch's directory control list. This is done by assigning a directory block, setting it with the arguments and linking it to the end of the current list. If the switch is "on", then the entry named by the sixth argument is located and removed from the branch's directory control list.

7. rename - This entry point is defined with the following CIMPL statement:

```
        rename: entry (dirptr pointer, name1(32) character, name2(32)
                   character, CODE integer);
```

The name of a branch in the directory pointed to by the first argument is to be changed, or two branches' names are to be switched. The branch list in the directory is searched for a branch with each given name. If neither or only the second is found a return with an error CODE is made. If only the first is found then the name in that branch is reset with the second given name. If both are found then the branch names in the two branches are switched.

8. list - This entry point is defined with the following CIMPL statement:

```
          list: entry (dirptr pointer, branch_name(32) character, target
                   pointer, CODE integer);
```

Two types of listing can occur. If the branch name given as the second argument is null then a list of all branch names in the pointed to directory and their respective type is made by copying the information into the following structure based on the pointer given as the third argument:

```
     declare 1 branch_names_list based,
               2 num_branches integer,
               2 branch(500),
                 3 name(32) character,
                 3 dir_sw bit;
```

The order is the same as that of the directory's branch list. If more than 500 branches are in the directory the first 500 are listed.

The second kind of listing occurs if the given branch name identifies a branch on the directory's branch list. In this case the contents of the branch are listed in the following structure which is based with the third argument.

```
     declare 1 branch_cnts based,
               2 name(32) character,
               2 dir_sw bit,
               2 length integer,
               2 num_users integer,
               2 num_ctl_list entries integer,
               2 ctl_list(500),
                 3 user_name(32) character,
                 3 indicator(11) bit;
```

The branch name and directory switch are obtained from the branch itself. If it describes a directory segment then the control list given is the directory control list, also from the branch. Only the last four bits of the indicator in each structure entry are used to contain each four bit directory access indicator. The remaining needed information is obtained from a call to the list entry in the segment catalog manager. (The returned access control list with one entry is ignored.) If the branch describes a non-directory then the access control list is instead listed in the structure.

This, along with the segments length and number of current user is obtained from the list entry in the segment catalog manager. If the control list is longer than 500 entries only the first 500 are listed.

Error codes

The six calls to the directory manipulator from the hierarchy access validator includes as a return argument an error CODE. The following table specifies the meanings associated with its possible values, and indicates which values can be returned from which entry points.

| CODE | can be returned from entry points | | | | | | meaning |
|---|---|---|---|---|---|---|---|
| | 3 | 4 | 5 | 6 | 7 | 8 | |
| 0 | * | * | * | * | * | * | request successfully performed |
| 1 | | * | * | * | * | * | named branch not in directory |
| 2 | * | | | | | | branch with given name already exists |
| 3 | | * | | | | | attempt made to alter length of directory |
| 4 | | | * | | | | attempt made to delete directory with branches |
| 5 | * | | | * | | | directory full, no more room |

## Identification

Segment locator

## Purpose

The segment locator is responsible for locating a segment specified by tree name within the SMS hierarchy, and obtaining a (segment number|word number) pointer that will allow a process to reference the segment. It is called by the hierarchy access validator to obtain pointers to directory segments about to be referenced by the directory manipulator and by the search director to obtain pointers to non-directory segments for the use of user procedures outside the SMS. The directory manipulator is called to locate branches within given directories and produce pointers to the segment corresponding to a specific branch.

## Introduction

The algorithm used to convert a given tree name into a pointer to the corresponding segment is described generally in the notebook section on sharing and protection of segments. The first step is always obtaining a pointer to the "root" directory. Because of its special nature (it can have no branch in the hierarchy) a special means is used to obtain it. The pointer needed is written into the code of this procedure as a constant, for each process' address space is initialized containing the "root" as referencible by the same segment number. Also written in to the code is the single directory control list entry for the "root" directory.

The segment locator is always careful to lock directories before passing a pointer to them on to the directory manipulator to have the directory searched for a specific branch. The locking strategy employed eliminates the possibility of two processes waiting for locks each other have set, a situation known as the deadly embrace. The strategy is as follows:

1. A lock is set by calling the lock entry point in the locker.

2. After obtaining a pointer to "root" it is locked before proceeding further.

3. Each time that a pointer to the next directory in a path is made available this directory is locked before the lock on the directory containing its branch is reset with a call to the unlock entry point for the lock.

4. If the segment located is a directory, to be referenced by the SMS, it remains locked after being located until all manipulations are completed. (The lock will be reset by the hierarchy access validator.)

## Entry points

Following is a description of the two entry points in the segment locator. The first is called by the hierarchy access validator, while the second is called by the search director. The procedure segment is named "seg_loc".

1. get_dir - This entry point is defined with the following CIMPL statement:

```
        get_dir: entry (dir_t_name(320) character, DAI(4) bit,
                 NO_DAI_SW bit, PTR pointer)
```

The first argument is the tree name of the directory to which a pointer is desired. The representation is the normal one with periods separating components. The name ends with the null character.

A pointer to "root" is obtained and its directory lock set. If the given tree name has only one component then this pointer is returned with the directory access indicator from the "root" directory's single entry access control list. If the given tree name has more than a single component then the following steps are taken for each additional component, in order to their appearance. The pointer to "root" is the initial current directory pointer.

a) call the get_dir_ptr entry point in the directory manipulator with the current directory pointer and the next component name as arguments.

b) If a null pointer is returned unlock the current directory and return to the caller with a null pointer.

c) Otherwise lock the directory to which a pointer is returned and then unlock the current directory.

d) Make the newly locked directory the current directory.

When the four steps have been completed for all componenets in the given tree name then the current directory pointer is the directory pointer desired. The directory identifier, the directory access indicator, and the no DAI switch returned from the last get_dir_ptr call are returned to the segment locator's caller along with that pointer.

2. get_nondir - This entry point is defined with the following CIMPL statement:

```
        get_nondir: entry (nondir_t_name(320) character, val_lvl
                    integer, PTR pointer);
```

This entry point is very similar to that described above. The first step, in fact, is to call that entry point with the tree name given as the first argument minus the last component, to obtain a pointer to the directory segment containing the branch for the desired non-directory. If a null pointer results it is returned to the caller. Otherwise the resulting pointer is supplied as an argument to the get_nondir_ptr entry point in the directory manipulator along with the branch name that is the last component of the given tree name and the validation level given as the second argument. The result of this call, a pointer, passed back to the segment locator's caller, after unlocking the final directory.

## Identification

Hierarchy access validator

## Purpose

The hierarchy access validator, part of the hierarchy control portion of the SMS, receives calls from the user interface manager specifying manipulation of a specific directory within the SMS hierarchy. The directory is identified by complete tree name. Two operations are performed before the call is passed-on to the directory manipulator which actually performs the manipulations: the tree name is converted to a pointer to word zero of the directory, and the request is validated against the directory control list for the directory.

## Introduction

Six of the eight entry points to the directory manipulator of the SMS are for manipulating a directory segment. User procedures wishing to invoke these entries call instead the same-named entries in the user interface manager. After performing certain transformations on the arguments, the user interface manager passes the calls on to the same-named entry points in the hierarchy access validator. Included in the argument list in each case are the full tree name of the directory to be manipulated, and a validation level. (See the notebook section on the user interface manager for a discussion of the validation level.) The tree name is converted to a (segment number|word number) pointer to word zero of that directory by calling the get_dir entry point in the segment locator. Also returned is a directory access indicator. This is the contents of the entry on the directory control list for the directory which corresponds to the owner of the calling process. The validity of the request is determined by comparing the validation level argument with the directory access indicator. If valid then the call is passed-on to the directory manipulator.

## Validation

The validation comparison is very simple. If no directory access indicator is returned from the segment locator then it automatically fails, for the user is not on the directory's directory control list. If an indicator is returned, and the call is to the list entry point, T = 0 or T = 1 with the validation level of the caller less than or equal to V is necessary. A call to one of the other five entry points passes under almost the same conditions, except only T = 1 is allowed. (T and V are the two directory access indicator fields.)

## Control list validation

One entry points allows users to specify access or directory control list entries. A call intended for the directory manipulator entry point change_ctl_list can include in its argument list an access mode indicator or a directory access indicator (one or the other, depending on whether the segment is a non-directory or a directory). The hierarchy access validator checks all ring numbers contained in the indicator. If any are found to be less than the validation level of the caller then they are set equal to it. This prevents , for example, a user from creating a procedure which he may call an have execute in ring zero. (The ring numbers in an access mode indicator are the R1, R2 and R3 fields, and in a directory access indicator the V field.)

## Entry point list

For the sake of completeness the CIMPL statements defining the entry points to the hierarchy access validator are given. Each causes the call to be passed on to the same named entry in the directory manipulator. The procedure segment is named "ha_val". The argument list for each appears complex, but is merely the same as the list for the corresponding directory manipulator entry point with the directory pointer replaced by the directory's tree name, and the addition of a validation level argument. (See the section on the segment locator for an explanation of how tree names are packed into the 320 character array.)

```
create_seg: entry (dir_t_name(320) character, branch_name(32) character,
        length integer, val_lvl integer, CODE integer);

change_seg_length: entry (dir_t_name(320) character, branch_name(32)
        character, new_length integer val_lvl integer, CODE integer);

delete_seg:  (dir_t_name(320) character, branch_name(32) character,
        val_lvl integer, CODE integer);

change_ctl_list:  entry (dir_t_name(320) character, branch_name(32)
        character, dir_sw bit, ins_del_sw bit, user_name(32) character,
        indicator(*) bit, val_lvl integer, CODE integer);

rename:  entry (dir_t_name(320) character name1(32) character, name2(32)
        character, val_lvl integer, CODE integer);

list:  entry (dir_t_name(320) character, branch_name(32) character, target
        pointer, val_lvl integer, CODE integer);
```

## Error codes

The error codes returned from the directory manipulator are passed back to the caller by the hierarchy access validator. In addition, the hierarchy access validator may generate the following two error codes:

| code | meaning |
|---|---|
| 6 | named directory could not be found in hierarchy |
| 7 | validation check failed |

## Identification

Search rules

## Purpose

The search rules segment of a process has two purposes: to define the working directory of a process, and to define the sequence of SMS directories to be searched when locating a non-directory segment to be included in the process' address space. It is referenced by the user interface manager, the search director, and certain modules from other subsystems in Clics.

## Discussion

The following declaration is used in the tree name former to reference the search rules segment. The segment has the branch name"srch_rules" and appears in a process' process directory. The pointer required to apply the based structure is obtained from a link snapped when the process is initialized.

```
declare 1 srch_rules based,
          2 wrk_dir(320) character,
          2 num_rules integer,
          2 rules(*),
            3 dir_t_name(320) character;
```

The name of the current working directory, and each of the directory names in the search rules, is represented in the normal manner for tree names, with components separated by periods. The null character ends each tree name if shorter than 320 characters. See the description of the user interface manager and the search directory for the use of the search rules segment.

The search rules and the tree name of a process' working directory are set by the login command.

## Identification

Search director

## Purpose

The search director has the single purpose of directing the search of the SMS hierarchy for a segment to which a pointer is desired when its complete tree name is not provided by the caller, the user interface manager (which received it from a user procedure). The search rules segment is used to provide directory tree names with which to complete the provided partial tree name. The segment locator is called to produce the pointer once a complete tree name is found.

## Entry point

The single entry point in this procedure segment (whose branch name is "srch_dir") is defined with the following CIMPL statement:

```
        get_nondir: entry (t_name(320) character, val_lvl integer, PTR
            pointer, FOUND_T_NAME(320) character);
```

The tree name given as the first argument is inspected. The representation is the normal one, with periods separating components and a null character ending the tree name. If it is mal-formed a null pointer is immediately returned. If it is complete (i.e. begins with "root.") then it is passed-on as the first argument to the same named entry point in the segment locator. (The given validation level is the second.) The returned pointer is passed back to the caller.

If the user has not specified the complete tree name of the segment desired then the search director can be of further assistance. The search rules are a list of the tree names of directory segments that can be used to complete partial tree names provided by the caller. For each search rule the directory tree name is appended to the left of the partial tree name provided, and the complete tree name thus formed is treated as described above. If the final result is a valid pointer to be returned to the user then the successful complete tree name is returned also. If the result is a null pointer than the next search rule is tried. If all search rules are exhausted before a referencable segment is found a null pointer is returned to the caller.

## Identification

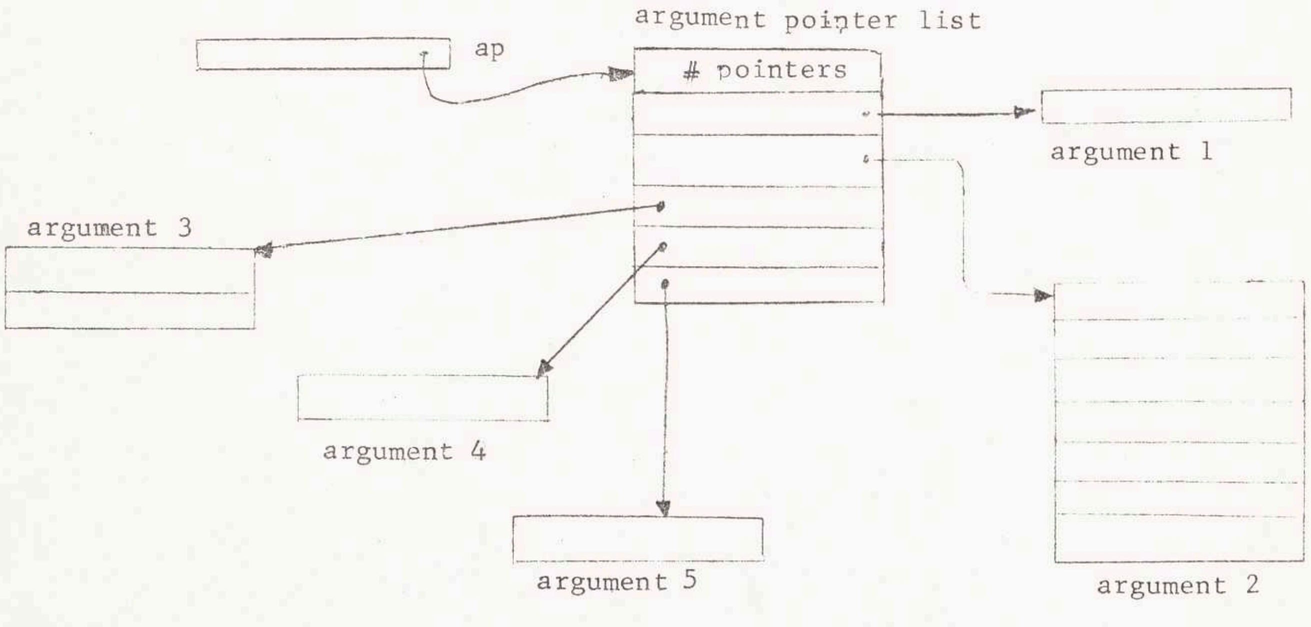
User interface manager

## Purpose

The entry points of the user interface manager are the only entry points in the entire SMS that may be called by procedures executing outside of ring zero. Thus, they provide the only direct user access to the facilities of the storage management subsystem. The function of the user interface manager is to validate, complete, and convert to the form used within the SMS arguments provided by procedures calling this subsystem. The calls are then passed-on to the relevant entry points in other SMS modules. Calls to the eight user interface manager entry points from outside of ring zero come via the ring zero entry vector.

## Introduction

A function common to the eight entry points to the user interface is argument validation. In the calling sequence constructed by the CIMPL compiler, ap (GPR5) is used as the argument pointer. It contains a (segment number| word number) pointer to a list of pointers which, in turn, point to the actual arguments.



If ap can be verified to point within a segment that is readable in the ring of the caller, then the protection mechanism built into the processor will cause a fault on reference if any member of the argument pointer list points to an argument not accessible from the caller's ring. Verification is accomplished by calling the get_ami entry point in the process address space manager, which returns the access mode indicator for the segment within which ap points. The R2 field of this indicator is then compared to the ring of the caller, which is found in word zero of the stack frame. If R2 $\geq$ ring of the caller then ap is acceptable. (If ap is found to be invalid, or if no access mode indicator at all is returned, then control is forced back to the listener in ring four by calling the appropriate entry point in the processor management subsystem.)

If it were acceptable to take an illegal procedure fault at any point of first reference to an argument within the SMS, nothing further would have to be done by way of argument validation. But such faults are not acceptable when a directory is locked on behalf of a process, for the directory would then remain locked forever. To prevents faults from occuring later, within other modules of the SMS, the user interface manager passes-on only arguments that are located in the ring zero stack. This means that a set of automatic variables is declared that can be used as arguments to lower levels of the SMS, and user provided input arguments are copied into the corresponding automatic variables before the call is passed-on. When the called module returns to the user interface manager all output arguments are copied from the automatic variables into the user procedure return arguments. In the case of the list entry point, where one of the arguments is a pointer to the target area for the listing, a surrogate target area is provided as well. The automatic pointer points to this new target in the ring zero stack. On the return the contents of the surrogate target are copied into the user procedure provided target.

The entry points to the user interface manager can be divided into three groups: those eventually intended for the directory manipulator via the hierarchy access validator, that intended for the search director and that intended directly for the process address space manager. Each group is described below. The procedure segment is named "uface_mgr".

## The directory manipulation interface

Calls to six of the user interface manager entry points are passed-on to the same named entry points in the hierarchy access validator, which passes them, in turn, to the same-named entry point in the directory manipulator. The argument list for each matches that of the corresponding hierarchy access validator entry point.

Below are the CIMPL statements defining these six entry points:

```
    create_seg: entry (dir_tree_name (320) character, branch_name (32)
            character, length integer, validation_level
            integer, CODE integer);

    change_seg_length: entry (dir_tree_name (320) character, branch_name(32)
            character, new_length integer, validation_level integer,
            CODE integer);
```

```
delete_seg: entry (dir_tree_name(320) character, branch_name(32)
        character, validation_level integer, CODE integer);

change_ctl_list: entry (dir_tree_name(320) character, branch_name(32)
        character, dir_sw bit, insert_delete_sw bit, user_name(32)
        character, indicator(*) bit,  validation_level integer,
        CODE integer);

rename: entry (dir_tree_name(320) character, branch_name_1(32)
        character, branch_name_2(32) character, validation_level
        integer, CODE integer);

list: entry (dir_tree_name(320) character, branch_name(32) character,
        target pointer, validation_level integer, CODE integer);
```

The steps taken by the user interface manager in response to calls to these six entry points are the same except for the entry point in the hierarchy access validator that is, in turn, called. The first is to validate ap, and copy arguments provided by the user procedure into the corresponding automatic variables that will be passed on.

The second step is to complete the tree name of the directory involved which is given as the first argument. The user presents this tree name as a single array of ahcaracter. The representation used is the normal one, with periods separating components. It is considered to end with the null character. This tree name is inspected. If mal-formed (i.e. if some component is longer than 32 characters) an error CODE of 8 is directly returned to the caller. If its first five characters are "root." then the caller has provided the complete tree name. Otherwise, a complete tree name is made by appending to the left of that provided by the caller the tree name of the process' working directory as obtained from the search rules segment. (Note that if the caller intends to manpulate the working directory then only the character "." need be provided by it as the partial tree name.)

The third step is to check to validaty of the users provided validation level. The integer given is compared against the ring occupied by the calling procedure, as obtained from the zeroth word of the current stack frame. If the given value is greater than or equal to that ring number, and less than or equal to seven, it will be passed-on as is. Otherwise it is first reset to the ring number, or seven, respectively. Thus, the validation level argument as received by the hierarchy access validator is the ring number of the SMS caller, or any ring higher, if the caller wishes to represent himself as occupying a high ring than it in fact does.

The corresponding hierarchy access validator entry point is now called. Upon return, all output arguments are passed back to the caller by copying them from the temporaries to the output arguments provided by the caller.

## Interface to obtain pointers

The single call that is passed on to the search director allows a user procedure to obtain a (segment number|word number) pointer to a non-directory segment. The entry point is defined with the following CIMPL statement.

```
        get_nondir: entry (t_name(320) character, validation level integer,
            PTR pointer, FOUND_T_NAME(320) character);
```

Again, the first step is to validate ap and copy the input arguments into the automatic variables for the call to the search directory. The validation level argument is validated as described above. The call is then passed-on to the same-named entry point in the search director. Upon return the two output arguments are copied from their temporaries to the user procedure provided output arguments, and a return made to the caller.

## Interface to check access

The final user interface entry point allows a user procedure to check its permission to access a segment. It is defined with the following CIMPL statement:

```
     check_access: entry (ptr pointer, access(3) bit, RESULT bit);
```

The argument pointer register is validated, and then the get_ami entry point in the process address space manager is called to obtain the access mode indicator corresponding to the given pointer. The three bits of the second argument are interpreted to specify the access wished, indicating read, write, and call, respectively. If such access is allowed by the access mode indicator from the ring indicated as the caller's ring by word zero of the stack frame, a RESULT of "1"b is returned. If no access mode indicator is returned, or the returned indicator does not allow such access, then a RESULT of "0"b is instead returned.

## Identification

Overview of processor management subsystem

## Purpose

The four major functions of the processor management subsystem are to perform the multiplexing of processors among processes, to allow processes to intercommunicate, to provide the interface between processes and the hardware fault/interrupt mechanism, and to assign and release processes to and from system users. The heart of this subsystem is a group of modules named traffic control which transform a process among the states of running, ready, and blocked.

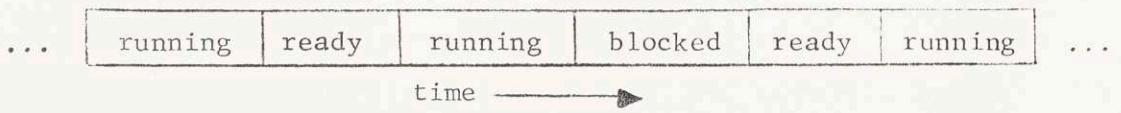
## Introduction

A process can be informally defined as a program in execution. Directly associated with each process in the Clics system is a virtual computer that consists of a pseudo-processor and an address space (containing procedure and data segments). Part of the task of the storage management subsystem is to implement the address space of each process. Part of the task of the processor management subsystem is to implement the pseudo-processor.

Implementation of each process' address space with a separate physical memory was rejected because it made meeting the system objective of information sharing very difficult, as well as because it was economically infeasible. On the other hand, implementation of each process' pseudo-processor with a dedicated Clics processor is rejected entirely for economic reasons. The technique of multiplexing a limited number of processors among an arbitrary number of processes makes fuller use of the capabilities of each processor, and thus more efficiently employs the resource of system processor capacity.

Processor multiplexing is done by the traffic control portion of the processor management subsystem. Traffic control is responsible for manipulating the state of each process within the Clics system. At any given moment, from the system's point of view each process is in one of the three states running, ready, or blocked. A running process is one that is executing on a Clics processor. A ready process would be running if a processor were available. A blocked process has no current use for a processor. It is waiting for something to happen - the arrival of a signal from elsewhere in the system.

To visualize multiplexing consider the progress of a process within the system. As time passes it switches back and forth among the blocked, ready, and running states as in the diagram below.



When it is blocked or ready it is not using a processor. The blocked state is entered by a process on its own initiative. When a running process goes blocked, some ready process goes to the running state and is assigned to the processor so vacated. On the other hand, a running process enters the ready state by directinn from traffic control, so that another ready process may run. Thus the processors are multiplexed among the processes by a combination of taking advantage of processes which go blocked of their own accord, and forcing those that do not to relinguish their processor now and then. The decision to switch a process to the ready state is made on the basis of a hardware interval timer that is set to some positive value when a process gains control of a processor, and that will interrupt the processor when it runs down to zero. The interrupt causes the process to place itself in the ready state by forcing it to call traffic control.

A primitive type of interprocess communication is also provided by traffic control. It is in the form of an entry that allows a process to cause another to be switched from the blocked to the ready state, and an entry that allows a process to force another back to command level. (See the notebook sections on the command subsystem.) Thus, a process about to go blocked arranges for another to signal it when a certain event occurs by calling traffic control to switch it back to the ready state so it can begin to run again when a processor becomes available. The second entry is used by a process to forward a quit signal received in the typewriter input stream attached to another process. (The input is not necessarily initially received by the attached process.)

A second area within the processor management subsystem, fault/interrupt control, provides the interface between the hardware fault/interrupt mechanism, and the processes within the system. Whenever a fault or an interrupt occurs in a processor, control is transfered to a location specified by a processor register. This location is within the fault/interrupt control portion of the processor management subsystem. Here the cause is determined, and appropriate action initiated. This means either translating an interrupt into a signal (via traffic control) for the interested process; or repairing, if possible, the cause of the fault.

The final area with the processor management subsystem is called process control. This portion performs the initialization necessary to assign a process to a Clics user. It also can release a process from a user. Process control, in addition, becomes involved in transfering the locus of control within a process to the command subsystem (i.e. forcing the process back to command level) in the event of an unrecoverable fault, a quit, or a logout.

## Process table

The primary data base of the processor management subsystem is the process table. This table contains a single entry for each process that is defined within the entire Clics system. The following information, among other things, appears in the process table entry for each process.

process identifier
owner's name
current state
quit switch
descriptor base register image
ring zero stack pointer
time limit
ready list link

The process identifier is a system-wide unique label generated when the corresponding process was built (at system initialization time) which identifies the process. Within the second item is recorded the name of the current owner of the process. (When the process is not assigned to a system user, the name ADMINISTRATOR appears here, assigning the process to the system administrator.) The third item records the immediate state of the process: running, ready, or blocked. The quit switch is used to transmit a quit message to the corresponding process.

The descriptor base register image is a bit string suitable for direct loading into the descriptor base register of a processor. When so loaded, it addresses the head of the descriptor segment of the process, allowing the address mapping logic of the processor to function within this process' address space. The act of loading it into a processor's descriptor base register starts the process executing on that processor, switching the process from the ready to the running state. The next item in a process table entry, a pointer to the current frame in the ring zero stack, is also loaded into a processor register at the same time.

Time limit is the value to be placed in the hardware interval timer when the process is placed in the running state. The final item, the ready list link, threads the entries of all processes that are in the ready state together into the ready list. Each time a processor becomes available, the process corresponding to the top entry on the ready list is switched to the running state by being placed in execution on that processor.

Modules of the processor management subsystem within each process have access to the entire process table via pointers to its zeroth word which are obtained from links snapped at system initialization time. In addition, other modules within each process have access to the specific process table entry corresponding to the process. The primary use of this access is for a process to determine the name of its current owner.

## Traffic control

There are four basic entries to the traffic control portion of the processor management subsystem. They are named block, wakeup, restart, and quit.

The entry named wakeup is used to switch an arbitrary process from the blocked state to the ready state, from which it will automatically switch to the running state when a processor becomes available. The argument is the identifier of the process to be awakened. When the call is made the current state word in the process table entry of the process to be awakened will indicate that it is running, ready, or blocked. If blocked, the state is changed to ready and the process is placed on the ready list to await its turn for a processor. If the process is already running or ready then nothing occurs.

The entry named block is called by a process that wishes to change its own state from running to blocked. It is the responsibility of a process about to call block to arrange for another process to wake it up some time in the future. The call causes the process' state to be changed from running to blocked, and the occupied processor to be assigned to the process whose process table entry is at the head of the ready list.

Restart is the entry called by a process when its interval timer runs out. It causes the process to be switched from the running to the ready state, inserted at an appropriate position in the ready list, and the process at the head of the ready list to be given the vacated processor.

The final entry, quit, allows a process receiving a quit character from a typewriter associated with some other process (the process that typewriter input is intended for is not necessarily the process that initially receives it) to pass the quit along to the other process. The identifier of the process to be quit is given. The result is the setting of a switch in the process' process table entry, which, when noticed, will cause that process to return control to command level.

## Fault/interrupt control

The fault/interrupt control portion of the processor management subsystem gains control automatically each time a hardware interrupt or fault is detected by the processor on which a process is running. If an interrupt is the case the input/output subsystem and the clock subsystem are each called to check certain data bases for the cause of the interrupt and translate it into wakeup or quit calls for the interested processes. While this occurs further interrupts are inhibited from occuring. At the conclusion of this operation control is returned to the instruction that was about to be executed when the interrupt occured. The total effect is to translate the hardware interrupt signal into a software signal(s) to affected processes.

If the case is instead a fault then fault/interrupt control determines the nature and location of the fault and calls the appropriate fault handling procedure. Some faults are unrecoverable. In these cases control is forced back to command level in the process via the process control portion of the processor management subsystem. The same occurs if potentially correctable faults are unrecoverable in a specific instance. In the case of a recovered fault control is returned to the faulting instruction once the recovery is effected. One of the faults processed is the timer runout fault which is caused when the hardware interval timer described earlier runs down. The procedure for handling this fault is simply to call the restart entry in traffic control.

## Process control

The process control portion of the processor management susbystem performs the logging-in and logging-out of a user to and from a process, and provides the mechanism through which control is forced back to command level in the cases of an unrecoverable fault, a quit, or a logout. Login involves initializing the process table entry with the new owner name, and making certain changes to the process directory's directory control list in the storage management subsystem, as well as alterations to the search rules segment, and other things. Logout is the reverse of these procedures accompanied by the cleaning-up of the address space of the process and the return of control to the command subsystem. This latter mechanism is used by traffic control to return control there in the case of a quit, and by fault/interrupt control in the case of an unrecoverable fault.

The notebook sections which are to follow this overview and which will detail the mechanisms within the processor management subsystem are not included in this version of the specification notebook.

## Identification

Overview of command subsystem

## Purpose

The command subsystem provides the mechanism through which a Clics user, seated at a typewriter terminal, directs the activities of the attached process. Inputs formatted in the command language are interpreted by this subsystem as directions to call the named entry points to procedures stored within the hierarchy of the storage management subsystem.

## Introduction

A process executing in the command subsystem is said to be at command level. A system user initially gains control of a process operating at command level. In this condition a process will accept and respond to messages in the command language that are typed on the attached typewriter terminal. The command language is very simple, both syntactically and semantically. Each such message is called a COMMAND. A COMMAND is defined as follows (using CIMPL syntax notation and notation variables):

```
     ENTRY [blank...CHARACTER-STRING]...↵
```

ENTRY is the name of an external entry point expressed as it would be in a CIMPL procedure calling it, and CHARACTER-STRING is any sequence of characters other than blank or new line ("↵"). A COMMAND is terminated by a new line character. For example, the line

```
     lister.printa  alpha.text ↵
```

is a syntatically correct COMMAND.

The command subsystem provides each COMMAND with the following semantics:

1. ENTRY is taken to be the name of an external entry point.

2. Each CHARACTER-STRING is taken to be a single argument of type character array.

3. A standard CIMPL calling sequence to the named ENTRY with the given character array arguments is constructed and executed.

Thus, the above example would be performed by the command subsystem by constructing and executing a call to the entry point "printa" in the procedure segment with branch name "lister", with the character array "alpha.text" as the call's single argument.

When that procedure completed execution, it would return to the command subsystem via the normal CIMPL return. The command subsystem would then wait for the next command from the user.

The two basic modules within the command subsystem, the listener and the shell, are now described briefly.

## The listener

The listener reads the typewriter input stream when a process is a command level, storing the characters received in a buffer. When the character "↵" is received the listener assumes the end of a COMMAND has occured, and calls the shell, providing the input buffer as an argument. The shell, after processing the command (see below) will return back to the listener which repeats its action after clearing the input buffer. Thus, the listener is nothing more than a read loop on the typewriter input stream connected to the process.

## The shell

The shell is almost as simple. It syntactically analyzes the input buffer message, and if it is a well-formed COMMAND, constructs and executes the call indicated. Constructing the call involves preparing an argument list to point to the given character array arguments and constructing a pointer to the named entry point with the aid of subroutines of the linker. The return from the called procedure is immediately followed by a return to the listener for the next COMMAND. IT should be noted that the procedure called in execution of the COMMAND is free to establish interactive communication with the user at the typewriter terminal. The listener is by no means the only procedure in a process that may read that typewriter.

The notebook sections which are to follow this overview and which will detail the mechanisms within the command subsystem are not included in this version of the specification notebook. Certain complications arise because the command subsystem is in ring 4 with user programs.

## Identification

System loading and initialization

## Purpose

System loading and initialization has the purpose of transforming a Clics hardware system with an empty memory into a operating Clics system capable of responding to user requests to login. The details of this transformation are not described in this section. Rather, the condition of the system immediately after loading and initialization have occured but before it is used is discussed.

## Discussion

A loaded and initialized system has a memory containing the information hierarchy of the storage management subsystem, and includes a group of processes. There is one process for each typewriter terminal data channel in the system's input/output controllers. Each process is attached (by an entry in a system-wide data base of the input/output subsystem) to one of these data channels, and is started executing in the listener of its command subsystem. The listener does a read of the data channel which causes the process to go blocked until a user dials-up and connects a typewriter terminal. (See the overview of the input/output subsystem.)

Included in each process are all ring zero and one control programs and data bases, plus the command subsystem in ring four. Also provided are procedure stacks for rings zero, one and four. A single physical copy of each procedure segment and system-wide data segment in this collection is shared by all processes. Each process is provided with its own copy of process-wide data bases such as the descriptor segment and the procedure stacks. Because the same collection of segments is initially made a part of the address space of each process, each is assigned the same segment number in every address space. When a single copy of a segment is shared among all processes then the corresponding segment descriptor word in each process points the same place. When each process has its own copy of a segment, the corresponding segment descriptor word in each process points to a different place.

Since the control programs in ring zero and ring one make no other use of the internal storage in their linkage segments than to store the outbound links for the procedure, the fact that the segments initially provided each process are referenced with the same segment numbers allows one copy of a linkage segment to be shared among all processes in the system rings. The outbound links are all snapped by the initialization mechanism.

Also part of the initialized system is a spare process for each hardware processor. This spare is assigned to execute on the processor when there are not enough user assigned processes in the ready or running states to occupy all processors in the system.

The processes described above are the only processes that ever exist in the system. When a user logs in to the system he is assigned the process that is logically attached to the typewriter data channel driving his terminal. When the user logs out then that process is freed, and can be assigned to the next user doing the same. All segments added to the address space of a user assigned process after the user has logged in are removed when he logs out. Thus each user is assigned a clean process.

## Identification

Overview of input/output subsystem

## Purpose

This overview provides a functional description of an interim input/output subsystem (IOS) that allows each process to read and write a typewriter terminal connected to a hardware data channel that is permanently logically associated with the process. Entry points that can be called by user procedures within the process are provided to read until a new line character is encountered and to write a specified number of characters.

## Introduction

The IOS is a group of procedures and data bases residing both in ring zero and ring one of each process in the Clics system. It has two entry point, read and write, that may be called by user procedures. The read entry point causes characters being typed on the typewriter terminal attached to a process to be collected into a user specified buffer. Return form this entry point occurs when a new line character is encountered. The write entry point causes the contents of a user provided buffer to be written on the typewriter. Both entry points, after initiating the requested I/O, cause the process to go blocked. When the I/O is finished the process is awakened and a return is made to the user procedure that called. The wait for the completion of the I/O, of course, is invisible to the user procedure.

## Entry points

In order to allow the use of the interim IOS in the absence of the notebook sections detailing its mechanisms, the definitions of the two user accessible entry points are presented in this overview. Both entry points are in ring one, and are called through the ring one entry vector. The containing procedure segment has the branch name "type_io".

1. read - This entry point is defined with the following CIMPL statement:

```
   read: entry(buffer_length integer, buffer(*) character);
```

The typewriter terminal attached to the process is read and the characters received are placed in the BUFFER array, beginning with its zeroth element. This continues until a new line character is read. When that occurs a return is made to the caller. Thus, the new line character is the last character placed in the BUFFER before the return is made. If the BUFFER is not long enough to contain all the input received before the break character is read, then only the first buffer_length characters are recorded there, although the return is still delayed until the break character is received.

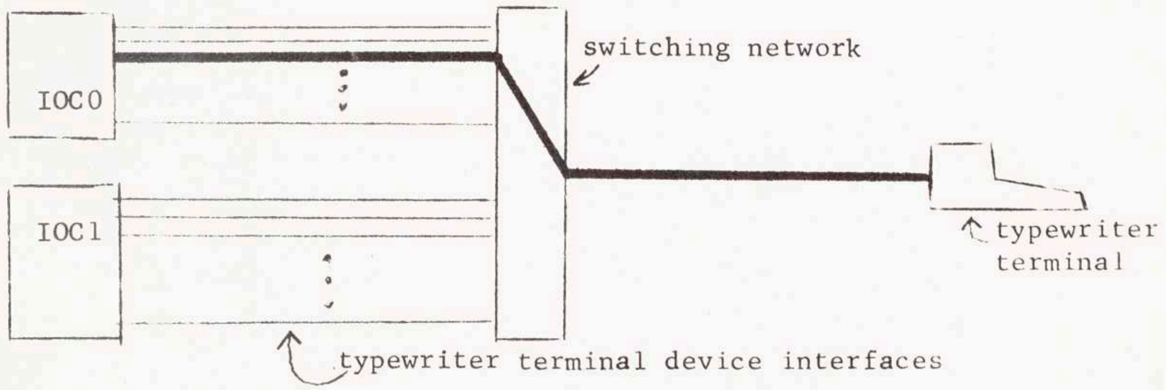
2. write - This entry point is defined with the following CIMPL statement:

```
write: entry(buffer_length integer, buffer(*) character);
```

The buffer_length number of characters from the buffer array are written on the attached typewriter terminal.

## Typewriter attachment

A process has no control over the typewriter terminal to which it is attached. At system initialization time each of 100 of the processes created with the system are attached to one of the typewriter terminal data channels of the input/output controllers. The attachment is logical and is accomplished by recording the relationship in a system-wide data base within the ring zero portion of the IOS. Thus, there is a one-to-one relationship between 100 processes in the system and the 100 typewriter terminal data channels. Each of these data channels is physically connected to an instance of the typewriter terminal device interface whose other end is connected to a telephone switching network. An individual wishing to use Clics simply dials the switching network (it has a telephone number) with his typewriter terminal's data set, and the switching network connects the line from his terminal to one of the typewriter terminal device interfaces leading to the ,system. The diagram below illustrates this method of connection.



Such a connection results in the logical attachment of the typewriter terminal to the process associated with the data channel connected to the selected typewriter terminal device interface. The attachment remains until the individual at the typewriter terminal hangs-up and the switching network breaks the connection.

When a process is attached in the manner described above to a typewriter terminal then it is assigned to the user seated at the terminal. A process not attached to a typewriter terminal is ownerless or idle. In order to become assigned again it must be aware that another typewriter terminal has become connected to the associated data channel. It is made aware in the following way. An idle process executes in the listener portion of the command subsystem which makes a read call to the IOS.

After the IOS has initiated this read it switches the process to the blocked state pending completion of the operation. As long as no terminal is connected to the attached data channel the read cannot be completed. When the switching network connects some terminal to the data channel, however, the terminal automatically sends the break character, completing the read. This causes the process to return to the running state and return control from the IOS to the listener which then initiates the login sequence. Thus, the ability of the IOS to cause the process to go blocked pending completion of an operation allows the processor to be used by another process with useful work to do. If this were not the case then an idle process could possibly loop for several days waiting for a dial-up, wasting much processor time.

## Implementation*

Part of the IOS is in ring one and part is in ring zero. User procedures call the ring one portion. On a read call the ring one portion passes the call to the ring zero portion which initiates the the operation and then returns control immediately to ring one. The ring one portion then goes blocked pending completion of the read. Incoming characters are placed in a special buffer associated with the hardware data channel by the data channel. When the break character is encountered the data channel signals the process via an interrupt which is translated into a wakeup by whatever process happens to be executing on the processor that receives the interrupt signal. When thus awakened, the IOS copies the received characters into the user defined buffer and returns.

On a write call the procedure is similar. The characters to be written are copied from the user provided buffer into the data channel associated buffer before being transmitted. A call to ring zero initiates the operation after which the IOS goes blocked in ring one pending completion.

The ring zero portion initiates reads and writes by constructing programs for the data channel to which the process is attached. The identity of this channel is obtained from the ring zero attachment table. The data channel is signaled to begin by sending an interrupt to the containing input/output controller. (See the notebook sections on the input/output controllers.)

The notebook sections which will follow this overview and detail the mechanisms of the IOS are not included in this version of the specification notebook.

*Note: It is not necessary to understand this subsection to use the IOS. It is given only to provide interested readers with the flavor of how the IOS works.

## Identification

Overview of dynamic linking

## Purpose

Dynamic linking is used to bind procedure segments to the procedure and data segments that they reference. This linking, which occurs at the time of first reference during execution, resolves the symbolic name of the external item that is part of the procedure segment making the reference into a two dimensional (segment number|word number) address.

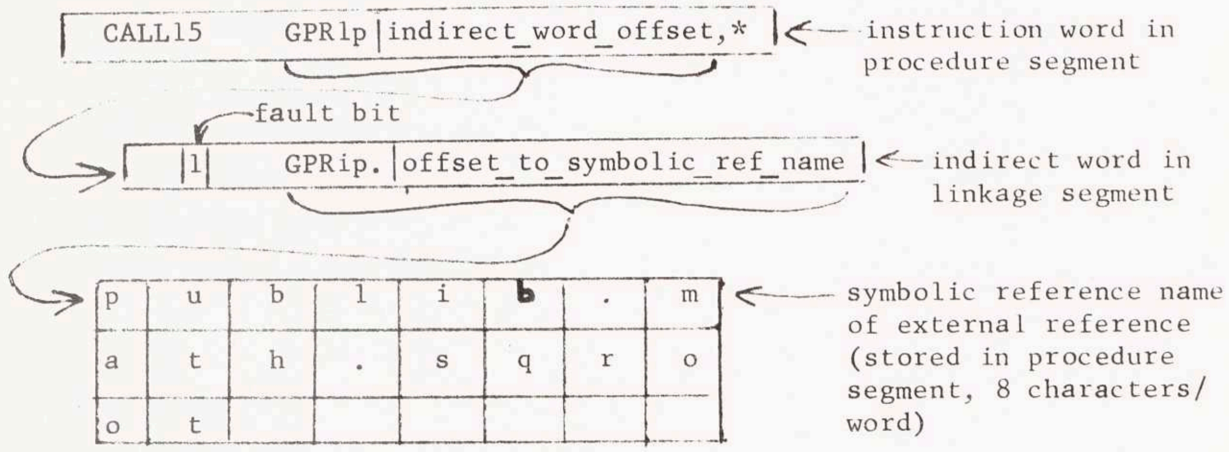
## Introduction

Symbolic procedures written in CIMPL, both system control programs and user programs, make two kinds of references to items that are neither in the procedure, its automatic storage, nor its internal storage: they call entry points in other programs and they reference externally defined data items. Call references are expressed by specifying the partial tree name of the procedure segment that contains the entry point and the name of the entry point. Data references are made with based structures applied to external data segments with pointer variables. The value of a pointer variable is expressed symbolically with a partial tree name and the name of a data item in that segment. (In Clics control programs such data references are always based at word zero of the external data segment. In such cases the data item name is omitted and only the partial tree name specified.)

## Compiler produced code for external references

The procedure segments for these programs produced by the CIMPL compiler maintain the symbolic form of external references. The code is designed so that when such a reference is actually made the dynamic linking mechanism is automatically invoked and a (segment number|word number) address is substituted for the symbolic link. Because the code produced is pure, however, this address cannot be substituted directly into the instruction making the reference. Putting the address in a register is also rejected because it would force the linking to occur again each time the reference was repeated. Instead, the address is placed in an indirect word which is not in the procedure segment but in its automatic or (most often) its internal storage. The internal storage is located in the linkage segment produced by the compiler as a companion to each procedure segment, and is referenced through a reserved general purpose register (GPRlp) which contains the address of its zeroth word. The compiler produces this indirect word with the contained fault bit set "on", and with its operand specification portion pointing to the symbolic name of the external reference. Thus, when the indirect word is used a fault is generated. This causes control to be given to the dynamic linking mechanism which replaces the faulting indirect word with one containing the address of the external item referenced.

A typical example is the code produced by the compiler for a call to an entry point not in the same procedure. This code is presented below:


CALL15 GPRlp|indirect_word_offset,*
instruction word in procedure segment
fault bit
1 GPRip.|offset_to_symbolic_ref_name
indirect word in linkage segment
p u b l i b . m
a t h . s q r o
o t
symbolic reference name of external reference (stored in procedure segment, 8 characters/ word)

(For an explanation of the representation used above for addresses see the notebook section on symbolic instruction representation.)

The call shown is to an entry point named "sqroot" in a segment with the partial tree name "publib.math". The indirect word is called an outbound link and is shown in the unsnapped condition the way that the compiler produced it. Dynamic linking causes the link to be snapped.

## Linkage segment

As stated above, the CIMPL compiler produces two segments for each program that it compiles; a pure procedure segment and a linkage segment containing the internal storage for the procedure - including the outbound links. In addition to the internal storage the linkage segment contains an inbound link and an entry code sequence for each entry point defined within the procedure segment. Both segments are stored in the hierarchy of the storage management subsystem (SMS) under tree names that are related. (See the notebook section on the organization of the SMS hierarchy for a description of the name relationship.) Each time that the procedure segment is included in some ring of some process a copy of the linkage segment is also included. This copy is altered by the procedure by storing internal variables into it and snapping the contained outbound links. When the procedure segment is removed from the address space the linkage segment copy is destroyed. The master linkage segment, the one produced by the compiler, remains unaltered as a source for new copies for other processes or other rings in the same process that wish to use the procedure segment.

## Linking

The dynamic linking mechanism can now be described. As the procedure segment containing the CALL15 instruction given in the example executes, GPRip contains the address of the instruction being executed and GPRlp contains the address of word zero of the associated linkage segment copy. When the CALL15 instruction is performed the processor will retrieve the indirect word addressed in the instruction word's operand specification portion (note that the address is complete) and evaluate its operand specification portion for the address of the location being called. The fault bit in this unsnapped link, however, will cause the processor to generate an indirect word fault. Control passes to the fault/interrupt interceptor procedure in the ring zero system. (See the notebook section on the fault/ interrupt mechansim of the processors.) This program, noting that the fault is caused by an indirect word, calls the linker, making available as input the saved registers of the faulting procedure. The image of GPRtp contains the address of the faulting indirect word, and that of GPRip contains the address of the (CALL15) instruction word.

The linker's task is to snap the unsnapped link and then cause the instruction to be re-executed. The snapped link will contain the address of the entry point being called, and the fault bit will be set "off".

Continuing with the example, then, the linker evaluates the address that is contained in the indirect word relative to the saved registers of the faulting procedure, and follows it to the symbolic name of the entry point being called. It considers the portion of this symbol array after the right-most period to be the name of the entry point within a procedure segment, and the remainder to be the partial tree name of the containing procedure segment. The next step is to obtain a pointer to that procedure segment. This is done by calling the "get_nondir" entry point of the storage management subsystem with the partial tree name as an argument. If no pointer is returned then nothing can be done and control is forced to command level through the processor management subsystem. If a pointer is returned, then also returned is the complete tree name of the segment found.

A linkage segmemt copy for the procedure segment must also be found. A transformation of the returned procedure segment tree name yields the tree name that the copied linkage segment associated with the procedure will have, if it exists yet. The "get_nondir" entry point in the SMS is called with this tree name to obtain a pointer to it. If no pointer results then a linkage segment copy is made by the linker from the master linkage segment in the SMS hierarchy. Once a copy is available the contained inbound links are searched for one containing the named entry point ("sqroot"). The matching inbound link will specify the offset in the linkage segment copy to the proper entry code sequence. (See the notebook section titled call - entry - save - return and procedure stacks for an explanation of the entry code in the linkage segment.) It is to this instruction sequence that the call is directed. So the original unsnapped link is replaced with a valid indirect word containing the linkage segment copy's segment number and the offset to the entry code sequence.

One final task remains. The last instruction of the entry code sequence is a transfer to the start of the code of the called entry point in the procedure segment. This instruction contains the proper offset, but does not contain the segment number of the procedure segment (it cannot be filled in a compile time). The linker inserts into this instruction the required segment number.

This completes the snapping of the link. Control is returned to the faulting CALL15 instruction and it is re-executed - without a fault this time, because the link has been snapped.

A simpler version of this occurs when the reference is to word zero of a data segment for the purpose of loading a pointer variable to base a structure. In such a case the symbolic external reference name ends with a period. All components are considered to be the partial tree name. Once the named segment is located nothing further need be done. The reference is to word zero of that segment, so the unsnapped link is filled in with the segment number and an offset of zero. It is not necessary to reference the data segment's linkage segment copy (if it has one).

## Linking of system procedures and data bases

In the case of system procedures in rings zero and one things are slightly different. Dynamic linking as described above does not occur. The compiler produced code is the same. But all links in these rings are snapped when a process is created. They cannot be allowed to be snapped dynamically because many of them are used in performing dynamic linking. The method used to snap them at process initialization time, however, is very similar to that described above.

The notebook sections that are to follow this overview and detail the format of linkage segments and the mechanisms of the linker are not included in this version of the specification notebook.

## Identification

Call-entry-save-return and procedure stacks

## Purpose

The five items named in the title of this notebook section work together to produce a completely recursive call and return facility that also provides the automatic storage used by a procedure. The ability of a call to cause an indirect word fault provides a trigger for the dynamic linking mechanism described in another notebook section.

## Introduction

Within a CIMPL procedure calls may be made to externally defined entry points. No restriction prohibiting closed loops of calls is present, implying that each CIMPL procedure is automatically recursive (i.e. may be called recursively). A new set of storage locations for all variables of class "automatic" is provided for each incarnation of a procedure.

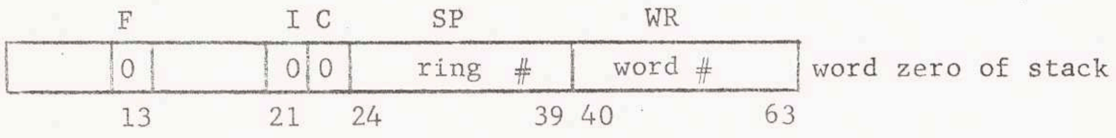
In order for a call to be returnable certain information about the state of the caller must be recorded. As a bare minimum, the instruction pointer must be incremented and saved so that, when returned to, the caller may continue to execute, beginning one instruction beyond the transfer of control. A more general mechanism is produced if all program accessable registers are saved. This is the case in Clics. The procedure ring register (PRR), the indicator register (IR), and the general purpose register 0 through 15 are saved. The latter group includes the instruction pointer (GPR0 or ip), the temporary pointer (GPR1 or tp), the call pointer(GPR2 or cp), the stack pointer (GPR3 or sp), the linkage pointer (GPR4 or lp) and the argument pointer (GPR5 or ap). In order for a call to be completely recursive a new storage area must be available to save the contents of these 18 registers each time a call is made. In fact, their stored image should be treated exactly as storage for variables of class "automatic".

To provide storage for both groups of automatic variable (i.e. both the variables of class "automatic" in a procedure, and the registers of the procedures' caller) a procedure stack is defined in each ring of each process. To make these stacks easily locatable by both program and hardware, they are the segments with segment numbers 0 through 7 in each process' address space, and the segment number designates the ring served. In use, the stack in a ring is pushed down to provide a new automatic storage area each time a call is made into or in the corresponding ring, and is popped up each time a return is made from a procedure in the ring (either to another in the same ring, or one in a higher ring). The area assigned to each incarnation of a procedure is called a stack frame, and a (segment number|word number) pointer to it is always available in sp when the procedure is executing. The first 18 words of a procedure's stack frame is used to save the machine conditions of its caller, and the following "n" words are used as storage of variables of class "automatic". The machine code associated with the call and return pushes down and pops up the stack.

Before considering in more detail the mechanism of the call and return, the format and operation of the procedure stacks is defined.

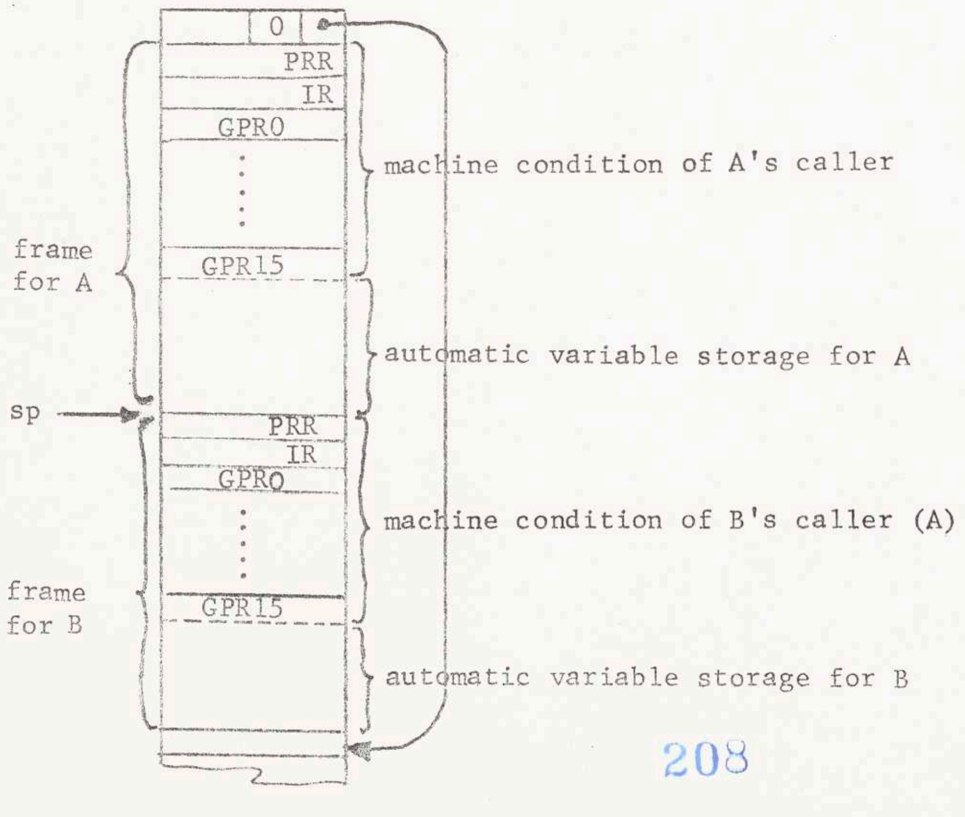
Procedure stacks

The procedure stack in each ring is a segment that may be read and written from that ring, but no ring higher. Thus, it is secure against violation by procedures in higher rings. The convention discribed above for assigning segment numbers to procedure stacks is known to the hardware, allowing it to select the stack to be used by any called procedure, once the ring in which the procedure will execute is known. Pushing and popping a stack is done by manipulating its zeroth word, which contains an indirect word pointing to the beginning of the unused area at the end of the stack. The format of this word is, then:



If the stack is completely empty, as would be the ring zero stack when a process is executing in ring four, for example, word # would be 1.

An incarnation of a procedure is assigned a frame on its ring's stack by loading sp from the zeroth word. The latter is then reset to point past the new frame (whose length will vary with the specific procedure). The following diagram illustrates a typical stack configuration. The ring zero stack is chosen as an example, and its state implies that some procedure outside of ring zero has called a procedure (A) in ring zero, which in turn has called another (B). Procedure B is currently executing.


0
PRR
IR
GPR0
GPR15
machine condition of A's caller
frame for A
automatic variable storage for A
sp
PRR
IR
GPR0
GPR15
machine condition of B's caller (A)
frame for B
automatic variable storage for B

When B returns to A the zeroth word of the stack will be reset from sp, and sp will be reset from the stored image of GPR3 in word 4 of B's frame.

## The call

A call statement in a CIMPL program (when the called entry is external) generates the following two machine instructions.

```
    EAPap      argpointer
    CALL15     GPR1p|link,*
```

The first loads the argument pointer register (GPR5) with the (segment number| word number) address of the argument list for the call, which has been previously constructed . (The present contents of ap are saved in an automatic variable before reloading ap if they are valuable.) The second instruction eventually results in reloading the instruction pointer GPR0) with the segment number and word number of the location called, thus causing control to transfer there on the next instruction cycle. The location to be called is addressed indirectly through an outbound link.
The indirect word in the link, initially contains an "on" fault bit.
Thus, when effective address formation references the indirect word a fault occurs. The fault handlers invokes the linker which replaces the faulting indirect word with the segment number and word number of the location to be called. This process, called dynamic linking, is fully described in the notebook section on the linker.

A description of the CALLn instruction appears in the notebook section on the processor instruction set. Some of that description is repeated here so that the operation in relation to the procedure stack can be understood.
The first step in performing the CALL15 that is the second instruction in the call sequence is to form the effective address of the location called.
In other words GPRsp| link,* is resolved to the segment number and word number of the called entry. The R2 ring number of the segment containing that entry is also determined. The ring in which it will execute is the minimum of R2 and the PRR. The PRR, IR, GPR0+1, and GPR1 through GPR15 are then stored in the beginning of the new frame in the stack of that ring. This is addressed as ring#|0,*. Note that GPR0, the instruction pointer is increment so that a return will come to the instruction following the call. sp is then set with a pointer to the new stack frame. Finally, GPR0 (ip) is set with the segment number and word number of the entry.

This completes the call. The next instruction executed will be the first at the entry. All machine conditions have been saved, and sp has been reset. Note, however, that lp and word zero of the caller's stack have not yet been updated. This is the functions of entry and save.

## The entry

The entry called is actually in the linkage segment associated with the

the procedure segment being given control. It consists of two instructions.

```
EAPlp      GPRip.|0
EAPip      proseg|save_offset
```

The first loads lp, the linkage pointer, with the (segment number|word number) address of the zeroth location of the called procedure segments' linkage segment. (Since execution is occuring in it, ip contains the proper segment number.) The second transfers control to the actual procedure segment, at the save sequence for the called entry. The segment number in the operand specification of this instruction was set by the linker.

## The save

Finally,control is in the procedure segment containing the program whose execution was desired. The last housekeeping chore is performed by the save sequence, which is always the first two instructions executed there. They are:

```
EAP6    GPRsp|18
ST06    GPRsp.|0
```

The cummulative effect of these is to increment the word number portion of the zeroth word of the stack by 18, causing it to point beyond the storage for the caller's machine conditions. If a larger frame is needed to include space for automatic variables then sp.|0 is further incremented by the code of the procedure.

## The return

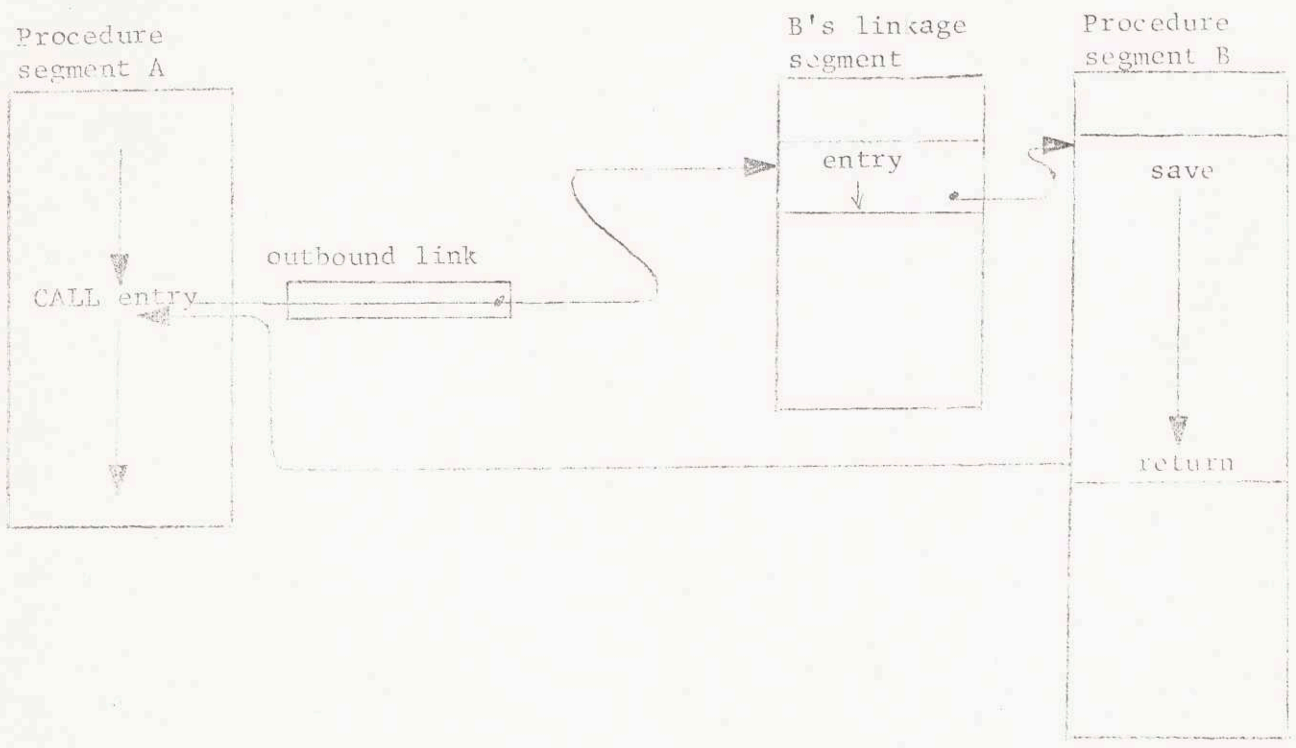
When the caller has finished execution it returns to its caller with the following return sequence.

```
STØsp    GPRsp.|0
RMC15    GPRsp|0
```

The first instruction returns its stack frame by resetting the zeroth word of the stack with the current stack pointer. The second resets all program accessable registers from their stored image in the first 18 words of the frame. Since this includes ip
its original state.

## In summary

The following diagram summarizes the interaction of call, entry, save, and return on a call from procedure A to an entry in procedure B.


Procedure segment A
CALL entry
outbound link
B's linkage segment
entry
Procedure segment B
save
return

## Identification

Entry vectors

## Purpose

Entry vectors are used in rings zero and one to prevent user procedures from calling locations within user callable procedures that are not entry points.

## Introduction

There are many entry points in rings zero and one that are intended to be accessible to user procedures outside of these system rings. If, however, the procedure segments containing them were made callable by giving call permission from all rings to the associated linkage segment copies (the call actually goes to the linkage segment) then calls could be directed to any word in these linkage segments - not just to the entry sequences. To prevent this for occuring, call permission is given only from within the system rings and a special segment, the entry vector, is defined for each of these rings. The entry vector is the only segment that may be called from outside of the system rings. Each word of the entry vector is a transfer (EAPip) instruction which passes control to one of the user accessible entry points in the ring. Thus, any word in this special segment may properly be called.

## Form of an entry vector

The entry vectors are built during system initialization. Each is a list of EAPip instructions whose operand specifications are the (segment number| word number) addresses of entry points within the ring that may be called from all rings. The addresses can be provided during initialization because system rings are pre-linked and the segment numbers of control procedures and system data bases do not vary between processes.

Associated with each entry vector is a linkage segment containing a separate inbound link for each EAPip instruction. The inbound link specifies a symbolic name formed by attaching the entry point name to the right of the branch name of the containing procedure segment with the character "_". The offset in each inbound link indicates the location of the corresponding transfer instruction in the entry vector. The branch names of the entry vectors themselves are "ring0" and "ring1". Thus, to call the "get_nondir"entry point in the user interface manager (branch name "uface_mgr") of the storage management subsystem, for example, a user procedure would direct the call to the entry point named "uface_mgr_get_nondir" in the procedure segment with partial tree name "ring0".

A later version of this section will describe in more detail the construction of entry vectors and a mechanism with which users may create them for the user rings.

## Identification

Locker

## Purpose

The locker, a ring zero procedure, is used by other procedures within the ring zero system to set and reset locks on system-wide data bases. If the specified lock cannot be set on first try then the locker causes the requesting process to go blocked until the lock has been reset. This allows other processes to use the processor while waiting on the lock.

## Introduction

The basis of the locker is the machine instruction STOZ, Store if operand zero. This instruction utilizes the READ AND HOLD memory command to perform the initial read of its operand, thus preventing other processors from accessing the operand until a second WRITE reference is made to it. Execution of an STOZ instruction by a processor involves the following steps:

1. Retrieve operand word from memory using the READ AND HOLD memory command.

2. Test operand for all zeros. If found then do step 3. Otherwise do step 4.

3. Store the contents of the instruction specified GPR in the operand using the WRITE memory command. Turn the zero indicator "on". Execution of instruction is finished.

4. Restore operand as read back into memory using the WRITE memory command. Turn the zero indicator "off". Execution of the instruction is complete.

A successful STOZ stores the contents (non-zero) of a GPR into the operand, preventing STOZ's performed by other processes on the same word from being successful. If the operand is a lock word in some system-wide data base, then a successful STOZ constitutes setting the lock. It is reset by storing all zeros into the lock.

## Entry points

The two entry points to the locker are defined below. The procedure segment has the branch name "locker".

1. lock - This entry point is defined with the following CIMPL statement:

```
lock: entry(lock integer, key integer);
```

The first argument is a lock variable from the data base to be locked. The second is the lock constant to be stored in the lock, usually the process identification integer of the calling process. The locker performs an STOZ of the key into the lock. If this is successful then it returns control to the caller. If it is unsuccessful then the identifier of the calling process is determined from the process table in the processor management subsystem and this identifier is entered in a system-wide locker table. The locker then calls the block entry point in the traffic control portion of the processor management subsystem, causing the calling process to go blocked. Upon return from the block entry point (which occurs following a wakeup) the steps described above are repeated.

2. unlock - This entry point is defined with the following CIMPL statement:

```
unlock: entry(lock integer);
```

The argument specifies a lock in some system-wide data base previously set by the calling process with a call to the lock entry point above. The lock is set to zero an a wakeup is sent to all processes listed in the locker table. Each time a listed process is awakened the corresponding table entry is removed. Then a return is made to the caller.

This section of the notebook will be expanded in later versions to provide a more detailed description of the locker, including the format of the locker table.

## REFERENCES

Abbreviations used in the references:

| | |
|---|---|
| AFIPS | American Federation of Information Processing Societies |
| FJCC | Fall Joint Computer Conference |
| ACM | Association for Computing Machinery |

References, in order cited:

[1] The Massachusetts Institute of Technology Bulletin, Vol. 103, No. 6 (July 1968), p. 268.

[2] Saltzer, J. H., "Introduction", Lecture notes for M.I.T. subject 6.233, Spring 1968.

[3] Corbató, F. J., Poduska, J. W., and Saltzer, J. H., Advanced Computer Programming, M.I.T. Press, Cambridge, 1963.

[4] Graham, R. H., "The Instran Compiler", Lecture notes for M.I.T. subject 6.251, Spring 1966.

[5] School Mathematics Study Group, Algorithms, Computations, and Mathematics, Stanford University, 1966.

[6] Sherman, P. M., Programming and Coding Digital Computers, Wiley, New York, 1965.

[7] Crisman, P. A., editor, The Compatible Time-Sharing System: A Programmer's Guide, second edition, M.I.T. Press, 1965.

[8] Multics System Programmer's Manual, Project MAC, M.I.T., to be published.

[9] Corbató, F. J., and Vyssotsky, V. A., "Introduction and Overview to the Multics System", AFIPS Conference Proceedings 27 (1965 FJCC), Spartan Books, Washington, D. C., 1965, pp. 185-196.

[10] Glaser, E. L., et al., "System Design of a Computer for Time-Sharing Application", AFIPS Conference Proceedings 27 (1965 FJCC), Spartan Books, Washington, D. C., 1965, pp. 197-202.

[11] Vyssotsky, V. A., et al., "Structure of the Multics Supervisor", AFIPS Conference Proceedings 27 (1965 FJCC), Spartan Books, Washington, D. C., 1965, pp. 203-212.

[12] Daley, R. C., and Neumann, P. G., "A General-Purpose File System for Secondary Storage", AFIPS Conference Proceedings 27 (1965 FJCC), Spartan Books, Washington, D. C., 1965, pp. 213-229.

[13] Ossanna, J. F., et al., "Communication and Input/Output Switching in a Multiplexed Computing System", AFIPS Conference Proceedings 27 (1965 FJCC), Spartan Books, Washington D. C., 1965, pp. 231-241.

[14] David, E. E., Jr., and Fano, R. M., "Some Thoughts About the Social Implications of Accessible Computing", AFIPS Conference Proceedings 27 (1965 FJCC), Spartan Books, Washington D. C., 1965, pp. 243-247.

[15] Daley, R. C., and Dennis, J. B., "Virtual Memory, Processes, and Sharing in MULTICS" Communications of the ACM, Vol. 11, No. 5 (May 1968), pp. 306-312.

[16] Saltzer, J. H., "Traffic Control in a Multiplexed Computer System", MAC-TR-30, Project MAC, M.I.T., 1966.

[17] Conti, C. J., et al., "Structural Aspects of the System/360 Model 85", IBM Systems Journal, Vol. 7, No. 1 (1968), pp. 2-29.

[18] Graham, R. M., "Protection in an Information Processing Utility", Communications of the ACM, Vol. 11, No. 5 (May 1968), pp. 365-369.

[19] Seligman, Lawrence, "No. 1 ESS Case Study", Lecture Notes for M.I.T. subject 6.233, Spring 1968.