HARDWARE SUPPORT FOR MODERN SOFTWARE CONCEPTS

Charles G. Hoch

August, 1978 ICSCA-CMP-12

ACKNOWLEDGMENTS

I am very grateful to Dr. James C. Browne for the immense amount of guidance and assistance he has provided me while I have been a graduate student. I am grateful to the Certifiable Minicomputer Project for providing financial support for this dissertation, and for committing itself to the implementation of Gypsy via an intermediate form based on the model described in this dissertation. Discussions of the model with Richard Cohen and Dr. Donald Good provided valuable insight. The work on the implementation of Gypsy with Larry Hunter and Larry Smith helped firm up some aspects of the model which had been vague. Thanks are due to Dr. James Peterson, who for his assistance in making the final revisions to the dissertation.

HARDWARE SUPPORT FOR MODERN SOFTWARE CONCEPTS

Publication No. _____

Charles George Hoch, Ph.D.
The University of Texas at Austin, 1978

Supervising Professor: James C. Browne

The gap between the abstract machines defined by modern software concepts and the abstract machines realized by most contemporary hardware is disconcertingly large. This dissertation describes and defines a realizable machine architecture which straightforwardly supports program modularity, typing and abstract data objects, concurrency, recursion, and program recovery/restart. The proposed architecture is obtained through a unifying synthesis and formalization of existing hardware concepts. Isolated examples of hardware implementations of many of the concepts now exist. Modern micro-electronics clearly permits the implementation of the unified conceptual base presented here. The key conceptual elements of the design are the integration of the concepts of typing, capabilities, word tagging, and memory virtualization, and the uniform treatment of hardware instructions, software functions, and

device actions.

The resulting architecture attains the goals of direct and effective support of modern software constructs and concepts while still retaining a simplicity which will allow implementation of high performance hardware at reasonable cost. The practicality of the architecture is demonstrated by using a software realization of it as an intermediate target language (abstract machine) for the implementation of Gypsy, a language which includes all of the concepts of modern software mentioned above. A compiler using this target language is being implemented.

Many if not most of the conceptual bases for this architectural design have appeared in the literature (often as software) in the past. The fundamental contribution of this dissertation is the integration of these concepts into a coherent computing structure and the demonstration of utility for such powerful abstract machines.

The fundamental conceptual structure used to define the proposed architecture is the integration of the concepts of capability based and tagged architectures. Everything represented in this system is an "object". Every object is referenced only through the use of a typed capability which we will label as a "descriptor". Even hardware instructions must obey this constraint. The architecture is built up by defining efficient forms of the

descriptor, using the type of the object, and defining functions for manipulating the descriptors, in order to implement the functionality of the architecture. An instruction call (hardware implemented routine), a procedure call (software implemented routine), and a hardware device call (such as a disk drive) are all represented as a descriptor for that "routine" (actually, a descriptor for an queue serviced by the routine) and a list of the descriptors which are to be its inputs and outputs (parameters).

TABLE OF CONTENTS

FIGURES

CHAPTER 1

INTRODUCTION

1.1 Overview Of The Architecture

The gap between the abstract machines defined by modern software concepts and the abstract machines realized by most contemporary hardware is disconcertingly large. This dissertation describes and defines a realizable machine architecture which straightforwardly supports program modularity, typing and abstract data objects, concurrency, recursion, and program recovery/restart. The proposed architecture is obtained through a unifying synthesis and formalization of existing hardware concepts. Isolated examples of hardware implementations of many of the concepts now exist. Modern micro-electronics clearly permits the implementation of the unified conceptual base presented here.

The key conceptual elements of the design are:

1. Integration of the concepts of typing [Hoare 72], capabilities [Fabry 74], word tagging [Feustal 73], and memory virtualization.

2. Uniform treatment of hardware instructions, software functions, and device actions.

The resulting architecture attains the goals of direct and effective support of modern software constructs and concepts while still retaining a simplicity which will allow implementation of high performance hardware at reasonable cost. The success of the architecture is demonstrated by using a software realization of it as an intermediate target language (abstract machine) for the implementation of Gypsy [Ambler 76], a language which includes all of the concepts of modern software mentioned above. A compiler using this target language is being implemented.

The well-informed reader will recognize that many if not most of the conceptual bases for this architectural design have appeared in the literature (often as software) in the past. The fundamental contribution of this dissertation is the integration of these concepts into a coherent computing structure and the demonstration of utility for such powerful abstract machines. Among the sources which have been most significant are HYDRA [Cohen 75], the Plessey 250 [England 72], the work of Dennis [Dennis 65], and papers on tagged architecture [Feustal 73].

1.2 Inadequacies Of Current Architectures For Modern Software

Modularization (that is, the decomposition of a program into units with rigorously defined interfaces) is a key concept of modern software design technology (see [Parnas 72]). Modularization implies the construction of rigorously defined domains of execution. Contemporary architectures generally provide only limited facilities to support such rigorous interfaces in executable memory. (There are exceptions, such as Multics [Saltzer 74]). A unified extension of protection through the reach of a domain and provisions for sharing are needed. Additionally, modularization creates a requirement for frequent and extensive context switching. This is a very tedious and expensive task on most contemporary architectures.

Data abstractions [Liskov 74] (that is, the hiding of the details of the representation of a logical data structure) is closely tied to modularization and the typing of objects. Contemporary architectures generally provide no support for type discriminations and make it very difficult to render mechanisms of implementation invisible. For example, an arbitrary set of machine instructions can be applied to the words which make up a data structure, treating the structure as an unstructured group of words.

Concurrency of task execution, both apparent and real, is a keystone of current operating system design and is an increasingly important element of application program specifications. Management

of concurrency requires both mechanisms for involuntary sharing of resources (such as the CPU) and for explicit cooperation through communications channels. Contemporary architectures generally provide no direct support for either mechanism (see [Ford 77] for some proposed mechanisms).

The ability to recover from errors and to restart an algorithm or procedure is another facility which is very difficult to implement on current architectures. This facility is needed to increase the reliability and fault tolerance of computing processes. See [Randell 75] for a proposed scheme for implementation of recovery/restart.

The deficiencies of contemporary architectures for supporting these facilities lead to complex compilers and elaborate and sometimes inefficient operating system implementations to support these fundamental concepts. This dissertation lays the design foundations for efficient implementation and execution of programs using these concepts.

1.3 Summary Of Proposed Architecture

The fundamental conceptual structure used to define the proposed architecture is the integration of the concepts of capability based [Fabry 74] and tagged [Feustal 73] architectures. Everything represented in this system is an "object". Every object

is referenced only through the use of a typed capability which we will label as a "descriptor". Even hardware routines must obey this constraint. The architecture is built up by:

1. Defining efficient forms of the descriptor, using the type of the object;
2. Defining functions for manipulating the descriptors, in order to implemement the functionality of the architecture.

The concept that all references must be through a descriptor leads directly to several of the significant elements of the architecture:

1. The unification of hardware and software functionality;
2. The integration of executable and device memories (such as disks) into an integrated and virtualizable structure.

An instruction call (a hardware implemented routine), a procedure call (a software implemented routine), and a hardware device call (such as a command to a disk drive) are all represented as descriptors for "routines" (more specifically, a descriptor for a queue serviced by the routine; see section 4.5.2) and a list of the descriptors which are to be its inputs and outputs (parameters).

It is straightforward to recognize that these concepts can be used to provide direct hardware implementation of modularity, data abstraction, concurrency, reliability, and recursion. It is not straightforward to formulate an integrated architecture which

efficiently and effectively implements these concepts. The balance of this dissertation details such an architecture and applies it to the definition of an intermediate language for the implementation of Gypsy.

CHAPTER 2

GOALS

There are a number of goals that should be considered in designing hardware architecture to support modern software concepts. This section details and discusses the software concepts as goals to be supported by a computer architecture. The language Gypsy [Ambler 76] will be used in chapter 4 as an example. This chapter discusses the concepts of modularity, abstract data objects, concurrency, and reliability. Under each concept, a brief description is given of how contemporary architectures address these problems, and how architectures could be designed to better support these concepts.

2.1 Modularity

Part of structured programming methodology [Hoare 72] is to divide a large programming task into smaller parts with limited and carefully defined interactions. Each of these parts (or modules) can be designed, implemented, and verified independently, given only the external specifications of the modules with which it must

interact, and constrained only in that it must satisfy its own external specifications. The rationale for this partitioning is that uncontrolled interactions very quickly take a program out of the reach of human comprehension, leading to confusion and errors, and permit minor errors to have widespread consequences. The enforcement of the boundaries between modules is closely related to the general problem of protection.

2.1.1 Contemporary Architecture -

On most contemporary machines, the user is given an amorphous block of storage in which to place the program's code and data. Some machines support hardware partitioning of the user's storage into "code only" and "data only" blocks (e.g., PDP11 [Digital 73], PDP10); a very few support definition of segments for user and operating system routines (e.g., MULTICS [Saltzer 74], Plessey 250).

In machines without hardware partitioning of a given user's memory space, the compiler is usually responsible for partitioning the code and data of the various routines, but once the linking loader has placed the program in memory, the hardware boundary between routines is lost; unless the compiler inserts validation code, there is nothing to prevent errors in one routine from causing a jump into the middle of another routine, access of the data of

another routine, or even the execution of data. In fact, programmers often use assembly language routines for the explicit purpose of circumventing modularity restrictions imposed by the compiler.

2.1.2 Better Hardware Support -

Modularity requires the convenient definition of complete execution domains and an efficient means for the creation and invocation of these domains (the "context switching problem"). Efficient and well defined communication (via shared data areas, for example) with other modules is essential and should be supported by the hardware.

2.2 Abstract Data Objects

Abstraction of logical structure from representation is another important modern software concept [Hoare 72]. Essentially, this means that all aspects of a computer system, from machine instructions to elaborate application packages, are viewed as data objects and routines which operate on those data objects. Essential to the concept of abstract objects is the hiding of the internal structure of an instance of an object from all but the routines explicitly defined to operate on it, but allowing other routines to

manipulate the instance as a whole (e.g., declare an instance of the object, or pass it as a parameter). Since only specified routines have access to the internal structure of a particular kind of object, implementation methods and proof of properties about the object depend only on those routines granted such access. The concept of data abstraction is closely related to the concept of modularity.

The properties of an abstract object are defined by its "type". Instances of the type are usually considered to be "variables". The type name is used, for example, to declare instances of an object, and in formal parameter lists of routines. It is essential to the concept of modularity that actual parameters are of the same type as the formal parameters, since properties of the parameters are determined from their type for use by the compiler and verifier.

Data abstraction can have a significant effect on a software system design. The implementation of an object can be changed without affecting users of the object. The software system simply provides abstract objects which users may use in building their own abstract objects. With the implementation-hiding property inherent in abstract data objects, the need for distinguishing between applications and operating system routines almost disappears. A user can be allowed to possess instances of data objects such as I/O devices, system schedulers, and system queues because the system

designers know that the user can only access such system objects via routines supplied by the designers.

2.2.1 Contemporary Architecture -

Most contemporary machines are based on the Von Neumann architecture. Among other properties, this implies that memory is a series of bits whose interpretation varies depending on the context in which they are used. The same bit string, for example, may be an instruction when analyzed by the instruction decoder, an integer if used in an integer operation, a floating point number when used in the floating arithmetic unit, or a character string in an I/O unit. A few machines use tags on memory words to denote which of the machine supported types is contained in the word, for example, the Burroughs B6500 [Hauck 68].

The compiler for such a machine is freed from certain type checking and, since the interpretation is contained in the word, is permitted to use general operations like "add" without having to specify "integer add" or "floating point add" [Feustal 73]. However, since most machines do not provide this facility, it is the software's responsibility to keep track of the type of the object in each word, generate the appropriate operation codes, and add validation code to prevent those erroneous uses of memory words which it cannot detect at compile time.

The monitor/user modes are the only levels of protection found on most existing machines. A program runs either in monitor mode, in which it can access the machine in essentially any way it wants, or it runs in user mode, in which it is restricted to operations which (hopefully) cannot affect the integrity of the operating system. The existence of only two levels of protection forces many operating system routines to use monitor mode, even though they only need slightly more privilege than user routines. There are thus large bodies of code in which minor errors can disrupt the entire operating system, making the problem of debugging an operating system very difficult.

In the Plessey System 250, one may have the right to execute a routine in a capability block, but not have the right to be able to read or otherwise access the capabilities in that block [England 72]. Part of the routine call mechanism is to give the called routine the access right which enables it to read (and thus access the blocks denoted by) the capabilities in that block, which name the storage blocks which contain the private data of an abstract object. "Classes" as defined in SIMULA [Dahl 66] are thus directly definable in the hardware. However, the notion of "type" does not exist in the Plessey system; there is no direct way in which a routine can validate that a capability it has is indeed the object it expects it to be. Also, the hardware does not support the definition of a routine which has access to the internal structure

of two or more abstract objects, even of the same type (such as concatenation of two strings).

2.2.2 Better Hardware Support -

To support abstract data objects, the hardware should provide mechanisms for defining abstract objects, and the access rights which a given routine has for specific typed objects, including its parameters. Efficient mechanisms should be provided for validating the types of the parameters, since unless the parameter lists of external routines are known at compile time, and linkage by the loader to precisely those routines is guaranteed at compile time, the compiler cannot enforce correct parameter types on calls to separately compiled routines.

Support for modularity implies the "concealment" of implementation and representation which is essential for supporting abstraction concepts, in that modules can only see the interface to other modules, not their representation.

2.3 Concurrency

Concurrent execution of several processes has been a part of computer architecture since the first time a central processor took an interrupt. The I/O device is a separate physical processor which

is given the ability to more or less demand the attention of the central processor at times of the device's choosing. Interrupt handling is usually designed so as to be transparent to the user task.

The concept of several asynchronous, regularly scheduled applications tasks simultaneously sharing a physical processor has existed for quite a few years, being implemented in multiprogramming operating systems (IBM System/360 [Amdahl 64] for example). However, provisions for several tasks executing (conceptually, at least) concurrently in an applications program is much newer. PL/I [IBM 68] contains a mechanism for forking off parallel tasks, using the TASK option on the CALL statement. Concurrent Pascal [Brinch Hansen 73] has processes. Gypsy [Ambler 76] permits concurrent execution of procedures.

Furthermore, multiprocessing is becoming a more seriously considered alternative; that is, several tasks of one or more applications programs simultaneously running on several physical processors, usually minicomputers. From the applications program's point of view, the number of physical processors should not be relevant; the system should act as if each of the program's concurrently executing tasks has a dedicated physical processor.

A blurring of the distinctions between application and operating system results from a functional view of the entire computer system. The concept of concurrency long present in

multiprogrammed operating systems is made available to all users of the computer system, and the formalization of these concepts in high level languages provides system designers with tools which simplify the design of multiprogrammed operating systems.

The separate tasks of a program usually need to communicate and synchronize themselves with each other. High level languages which deal with concurrency use several related techniques to accomplish this. The techniques include P and V operators [Dijkstra 68], critical regions [Dijkstra 68] [Brinch Hansen 70], message buffers [Dijkstra 68], and monitors [Hoare 74].

2.3.1 Contemporary Architecture -

A basic requirement for several tasks sharing one central processor is that the machine state when a task regains the processor is identical to the state when the task last lost the processor, so that the task is not aware that it was suspended. Often the machine state includes the contents of many processor registers. Notice the similarity between the need for context switches here and the need to support modularity and abstract data structures.

When the context change is due to an interrupt, it is usually the responsibility of the interrupt routine to insure this restoration of state. Since many interrupt routines are trivial

compared to the cost and time of moving the entire machine state to memory and back, the routine may only save and restore that part of the machine state which it must modify in order to accomplish its operation. Such an operation is moving a word between an I/O device data register and memory. Complications may occur if another interrupt occurs during this context saving/restoring procedure.

The PDP-11 [Digital 73] series alleviates this problem by including pushing the current program counter and program status word onto a runtime stack and loading them with values from standard locations in the interrupt handler as an indivisible part of the jump to the interrupt routine. The return from interrupt instruction includes restoring the program counter and status word from the stack as an indivisible operation. The treatment of interrupts is thus almost identical to the subroutine call and return instructions. The other processor registers, however, must be explicitly saved and restored by the programmer, as is necessary for subroutine calls.

Special I/O instructions and I/O interrupt handling contributes to making I/O software very inelegant. The PDP-11 series has attacked this problem by making the device status, command, and data registers part of addressable memory, and eliminating explicit I/O instructions. Each device has its own interrupt address, removing the need for polling the various devices upon receipt of an I/O interrupt. However, the values in the data

registers of the slower devices must be explicitly transferred to and from memory. Interrupts at the end of transfers by devices which transfer blocks of words directly to memory must be dealt with.

The CDC 6000 series does not have interrupts. Instead, the system consists of a central processor and several identical autonomous small computers called Peripheral Processors (PPs). Each of the PPs can directly access central memory and the hardware channels which are connected to controllers for the various I/O devices. Each PP also has a private memory. By consulting central memory locations, a PP receives requests for peripheral service from central processor programs. The PP can determine whether a channel is busy or idle, place information on or off the channel as required, and move the information to and from a user-accessible queue in central memory via the PP's private memory.

In the Plessey System 250 [England 72], processes are treated as abstract objects (a Plessey object is described in section 2.2.1). Processes ready for execution are placed on a ready queue, and whenever a processor is to service another process, a process manager routine executes on that processor (either as a consequence of being explicitly invoked by the process, such as a process synchronization routine requesting that the process be blocked, or implicitly through a processor trap, such as expiration of a timer). This routine takes the next process on the ready queue

and executes an instruction which cause the processor to resume execution of that process. The processor registers of the process losing the processor are stored in an area in that process object, and are loaded from an area in the process object gaining the processor. The process losing the processor is placed on an appropriate queue by the routine. Only the capability for the process object need be stored in the process queues.

The Plessey System 250 does not have interrupts. The trivial I/O operations (like putting the next word on the device data register and set the status word) are handled by peripheral interface units, which are special hardware devices which field interrupts from the external devices and put and get data and control information on hardware queues. These queues are emptied and filled by regularly scheduled system processes running on one or more identical main processor units. These processes are scheduled at sufficiently high priority to prevent the queues from overflowing. Thus instead of elaborate special purpose controllers, System 250 lets the general purpose processors handle these tasks.

Very few architectures support the concept of concurrency beyond the interrupt handling mentioned above; operating systems implement multiprogramming and compilers implement multitasking in user jobs. In the CDC 6000 series, there is a very useful instruction (called an exchange jump) which exchanges all central processor registers with a specified block in central memory; the

instruction can be executed either as the result of an explicit instruction in a central processor program or as the result of a command given by a PP. All of the state information about a job is contained (after execution of the exchange jump instruction) in the user's memory area plus another block of memory called the control point area (CPA) which is not accessible to the user program. Thus it is easy to treat jobs like files for swapping or dumping to external storage for later execution. Subtasks of user jobs could be created by establishing "subcontrol points" using the exchange jump instruction.

The Plessey System 250 provides hiding of abstract object implementation, so processes can be treated like any other abstract object, and used by operating system and user alike. Thus process scheduler routines can move processes to and from various queues like any other data object. In particular, a "next-process" routine running on a given processor can examine a queue to determine the next process which needs processor service, and execute a "change process" instruction using a capability for that process. Thus, even the scheduler routines need not know how many processors are active.

2.3.2 Better Hardware Support -

To support concurrency, the hardware must provide an efficient, indivisible mechanism for changing the machine state of the central processor from one task to another. Provisions for handling multiple processors should be included, as should facilities for organizing concurrency into logical structures. Hardware to handle trivial operations of device handling would be useful because it would eliminate context changes to processes which execute only a couple of instructions. An efficient context switch mechanism would also solve this problem. The hardware should support the definition of processes as data objects, in order to facilitate both operating system management of user jobs and user defined subtasks of jobs.

Interrupts can be considered context switches to tasks (i.e., interrupt handlers) which are scheduled at high priority upon occurrence of an external event. Interrupt routines as special entities disappear.

2.4 Reliability

When a computer system is looked upon as a set of abstract objects and the routines which work on them, reliability becomes the ability of a routine, given that its entry specification is satisfied, either to return normally and satisfy its normal exit

specification, or to return an error state and satisfy an error exit specification, even if routines which that routine invokes fail to exit normally. Error detection thus occurs quickly, at routine call and return points.

Formal verification is promises to be a powerful technique to enhance reliability of software. A verification system sees essentially three types of specifications: (1) those axioms on which it is built and are assumed true, such as rules of inference and the semantics of the language; (2) those user defined specifications which it will prove using the rules of inference and the semantics; and (3) those user defined specifications which will be assumed in the static proof and validated at runtime. The user defined specifications define the semantics of the virtual machine or machines (that is, the abstract object and their routines) which the user has created for the problem domain. Runtime validation is used for those specifications not proven from more primitive abstractions. The semantics of the programming language define the virtual machine on which the programming language executes. The compiler can be used to "prove" some of the semantics of the language, such as name declaration rules, but must insert runtime checks for other rules, such as value checks for subrange variables.

In turn, the compiler makes use of primitive routines provided by the hardware and the operating system. Some aspects of the hardware routine can be statically verified, but the physical

properties of the hardware assumed in its design may change (e.g., a wire will come loose) in unpredictable ways. These anomalies must either not affect the functionality of the hardware unit, or must result in an error state, just as in software modules.

In order to detect failures in a software or hardware routine (other than those reported from called routines), there must be redundancy; that is, two or more representations among which a specified relation holds. For example, in a memory with even parity, "odd (N (x. data_bits)) iff (x. parity_bit = 1)," where N returns the number of one bits in its argument and x is a word in memory, must hold in the memory read function. Similarly, in a software example, a list may be represented as doubly linked, and the relation "NODE (NODE (x). forward_link). backward_link = x," where NODE returns the value indicated by its pointer argument, must hold on any reference to the list.

The user should not be burdened by having to explicitly take into account the redundancy every time such an object is used, since the redundancy is irrelevant to his use of the object. The concept of data abstraction addresses this problem by collecting all manipulations of the representation of an abstract object into several specific routines. The specification and manipulation of the (redundant) representation are localized and unknown outside of these routines, except for the error states which may be returned if the redundancy tests indicate inconsistency. The modular nature of

routines has the important effect of encapsulating errors inside the domain of the routine, thus limiting the amount of damage an error can do to how much damage loss of that module can do.

Redundancy may be adequate only to detect inconsistencies; more elaborate redundancy may permit restoration of a valid state without loss of information. An example of the latter is triple modular redundancy, sometimes used in fault tolerant hardware. Here, three modules in a unit compute the same function, and a majority output is taken.

The propagation of error states to calling routines has been mentioned before in this section. The philosophy concerning error states is to return the system to a normal (i.e., valid) state as soon as possible, with the minimum possible disruption to the system. This is the recovery phase of error handling. Errors are dealt with locally if possible; if not, the error is passed back to the calling routine, which may pass it back to its caller, and so on, until a normal state is achieved. Complete restart from read-only storage is the most drastic way to return the system to a valid state; that is, initialized.

Recovery involves data and control recovery. The former is determination of which variables have been corrupted and what values to give them to return to a valid state; the latter is the determination of where to resume execution. The recursive cache [Randell 75] is one mechanism to support restoration of a routine's

state to some previously attained valid state. Program modularity greatly helps in both the problem of data and control recovery, especially if modularity is enforced by the hardware. If a runtime error is detected, the recovery mechanism can be sure that only the module's local data and called-by-reference parameters can be invalid. Also, one can be reasonably sure that the processor was legitimately executing that module (e.g., it did not jump into the middle of the module from some other module) and therefore the point at which execution is to be resumed can be decided using the knowledge of the identity of the module in which the error occurred.

Assertions placed in the program which are denoted to be post-execution validated provide a mechanism for determining correct operation of the program after the fact, and are therefore useful in post mortems of a system crash. They also provide valuable raw data for performance analysis (for example, which routines are most used).

2.4.1 Contemporary Architecture -

The design and use of fault-tolerant hardware has been actively pursued in recent years, especially in military and space applications. Fault-tolerant hardware is designed to maintain its functionality in spite of faults which may occur in it. Such hardware usually includes fault indicator lines, some of which

signal faults which have been compensated for, and are used to assist repair personnel; other lines cause fault traps in the central processor. The Plessey System 250 [Repton 72] uses the latter as part of a telephone control system which has an availability requirement of no more than ten minutes down in fifty years.

The use of parity checked memory is common. The Plessey System 250 computes parity on both the address transfer to memory and the data transfer to and from memory [Hamer-Hodges 72]. The memory unit in the Texas Instruments ASC [Watson 72], as in many large machines, corrects one bit errors and detects two bit errors.

Due to the comparatively error-prone nature of magnetic tape transfer, most tape drives have for a long time included parity checking. Disk drives also often have parity checking.

None of the above hardware features aid in the runtime detection of errors in virtual machines defined in the software, other than those causing error states in the machine instruction (like divide by zero) or those caused directly by faults in the hardware itself.

2.4.2 Better Hardware Support -

Support of such concepts as modularity and functionality as discussed previously, help enforce routine calling protocols such as use of entry points and parameter passing conventions. The hardware should include provisions for the support of error returns from both hardware and software routines, and provide other support for recovery from runtime errors. Consistency checks for user defined objects should be supported, such as block checksums. The concept of overflow checks on arithmetic operations should be extended to support subrange restrictions.

A user invoked trace mechanism should be provided which does not significantly perturb operation of the system. The significance of being user invoked rather than a "snapshot" (such as a memory dump) of the system is that the former reflects events in the higher level virtual machine; piecing together the hardware events provided by the latter into what the higher level machine was doing is difficult. A device onto which routines can write trace information would be useful toward providing such a trace mechanism.

CHAPTER 3

CAPABILITIES AND TAGS

In this chapter, the utility of capabilities and tags is discussed, along with existing implementations of them.

3.1 Capabilities

In this section, the general advantages of a capability-based operating system are discussed, along with capabilities as implemented in the Plessey 250 [England 72] and the SRI Provably Secure Operating System [Neumann 75].

3.1.1 Advantages Of Hardware Capabilities -

Capabilities support the definition of software modules; if the capabilities are implemented in hardware, then the software concept of module can exist at runtime and can be enforced by the hardware at runtime, even against most failures in hardware instructions and addressing mechanisms. This module concept strongly supports the Gypsy concept of independent units.

Capabilities provide protection on the access path to an object. Possession of a capability is necessary and sufficient to permit the holder access to an object in a manner specified by the capability. Thus one routine may have a capability which grants read-only rights to an object, another routine may have a capability which grants read-write access to the same object. In order to validate every access to every object, efficiency dictates the use of hardware capabilities. This aspect of capabilities is discussed further in [Fabry 74].

A capability based operating system defined in hardware does not need a "privileged" mode of operation. Operating system routines have capabilities for objects like hardware I/O queues, physical memory, and user routines; not giving such capabilities to a user effectively reserves the manipulation of such objects to operating system routines. Capabilities in this way support the concept of functionality; that is, abstract objects and routines which operate on them. This functionality property in the architecture provides support for the implementation of data abstractions in Gypsy.

3.1.2 Capabilities And The Plessey System 250 -

System 250 [England 72] is a system designed for the British Post Office to control the telephone network. It is a multiple CPU, multiple store (memory) module system in which any processor may execute any task assigned to the system, and all store modules and I/O devices are accessible equally to all processors. The system hardware was designed along with the software with the goals of fast and reliable fault detection, effective fault diagnosis, recovery at any cost (using the least disruptive technique that works in the case at hand), and maintenance and expansion of the system without disrupting service.

Hardware capabilities are the mechanism by which System 250 detects erroneous accesses to store, whether caused by hardware or software error. A capability is an identifier for a unique store segment (block), and is created only by the store manager. The capability includes the access rights granted by this capability to the block, such as read and write, as discussed later. These store segments contain either data (which includes code) or capabilities. Machine instructions are provided which load CPU registers known as capability registers with capabilities from capability blocks just as other instructions load data into data registers from data blocks. The capability registers contain base/limit addresses and access rights, and all memory references, including instruction fetches and subroutine calls, are done via a capability register.

Note that capabilities are freely copiable, though unmodifiable.

The access rights which may be denoted by a capability are (1) read data, (2) write data, (3) execute data, (4) read capability, (5) write capability, and (6) enter capability. The data rights (1, 2, and 3) act as expected. Read and write capability rights load and store capabilities from and to capability segments, just as read and write data load and store data registers. If the write data right is present, the capability rights are not allowed; if this restriction were not enforced, one could write data into a segment and then load a capability register from that segment, compromising security. The enter capability right is the subroutine call mechanism, explained later.

System 250 is based on a hierarchy of abstract machines, each of which expands on the facilities provided by the abstract machine below it. The two basic managers are the store manager and the process manager. The store manager abstracts distinct store modules and backup store into store blocks. It provides blocks of store of desired length and access rights, upon demand of any program. It also deallocates the block when explicitly released by the program.

An execution of a program is a process. The process manager is concerned with the creation, scheduling, and synchronizing of processes. It abstracts processors into processes.

Other managers are the input/output, textfile, directory, job, and command interpreter managers. Applications programs use these managers and those of their own creation to perform their tasks.

The managers each provide an allocator subroutine which may be called to generate a new, unique abstract object, and returns to the caller a capability for that object. If the object is a store block (from the store manager), the capability allows machine instructions to operate directly on the object; e.g., write data into the block from a data register. If it is a more complex object, the returned capability has the "enter" right for a capability block. This block (called the main capability block) contains execute capabilities for the code blocks which may operate on this abstract object, and capabilities for data and capability blocks which form the internal structure of the abstract object.

The CALL instruction is the mechanism by which access to the internal structure of an abstract object is controlled (see figure 1). The possessor of an enter capability for a block cannot read the capabilities contained in that block. The CALL instruction includes an enter capability for a main capability block and an index into that block which denotes an execute capability for the code block of the called routine. The effect of the CALL instruction is to push the current values of the two capability registers which hold capabilities for the calling routine's code

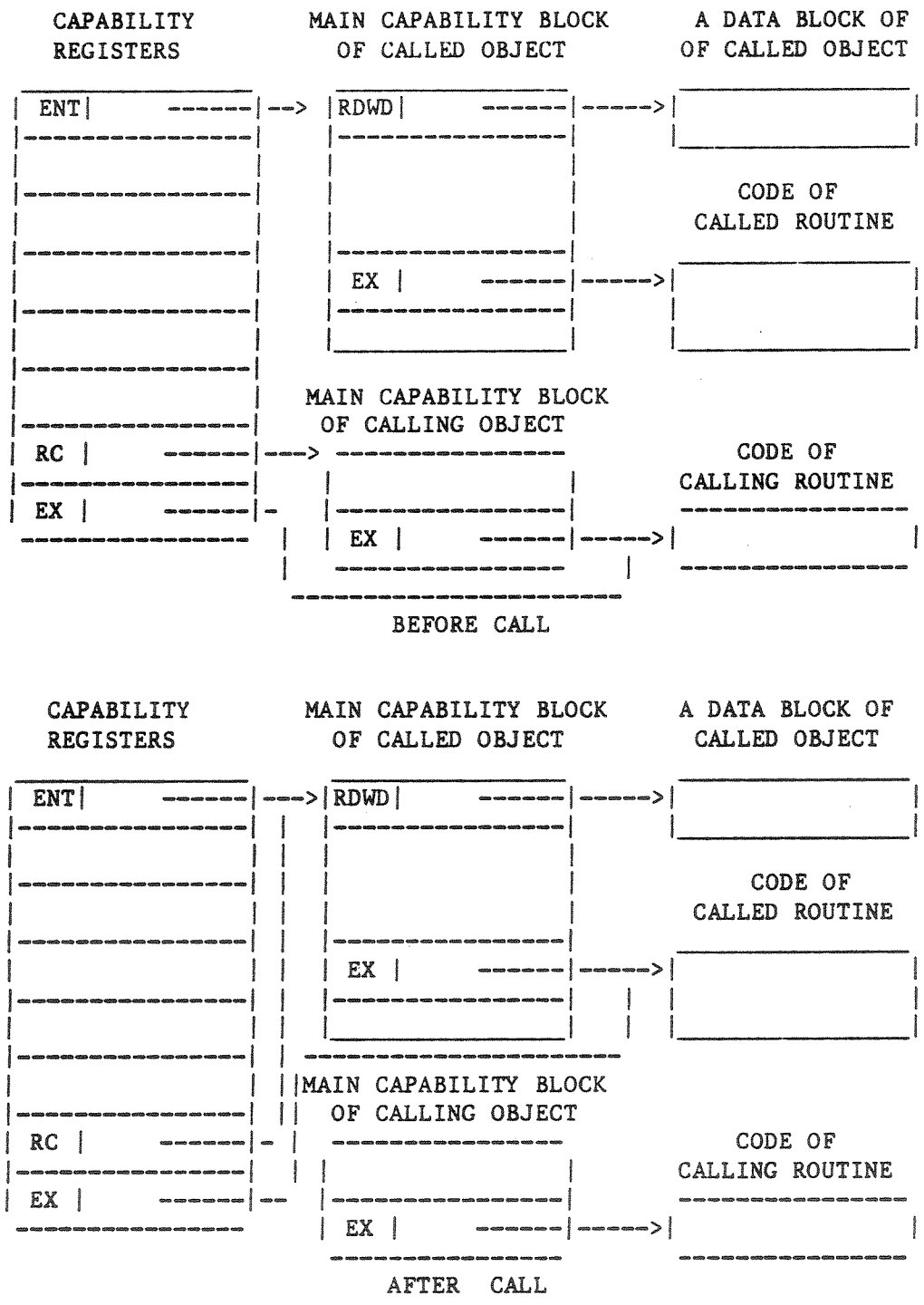


Figure 1. Plessey 250 ENTER operation

block and main capability block onto a stack associated with the process (not accessible to the calling or called routine), and load the registers with the denoted execute capability for the called routine's code block and a "read capability" capability for the called routine's main capability block. Thus the called routine can access the capabilities which form the internal structure of the abstract object. The calling and called routine are thus protected from each other.

The operating system is thus a set of subroutines and processes which look like and are used like any user defined subroutines and processes. Their only distinction is that they were created by the system staff and have jealously guarded capabilities for certain areas of store, such as those segments which are really hardware I/O queues.

Capabilities play an important part in limiting the scope of store corruption caused by faults, and in recovery as discussed in section 3.3.

3.1.3 Capabilities And The SRI Provably Secure Operating System -

In this system [Neumann 75], a capability is a two-tuple [uid, access-vector], where uid is a system wide unique name (identifier) for the object, and the access vector is a boolean n-tuple, where each position corresponds to an operation on that

object. Capabilities may be interspersed with data in an object, and are distinguished by hardware tags. The type of an abstract object is determined from the uid by a map in the extended-type manager, a system-provided module for user-defined abstract objects and system modules above the segment manager. The capabilities for the internal representation of objects of a given abstract type are contained in an object of type "TYPE", which is managed by the extended type manager. Operations on a given type possess the capability which permits them to obtain the representation capabilities from the extended type manager. The uid is the value of a system clock which is not permitted to wrap around.

3.2 Tags

In this section, the advantages of hardware tags are discussed, along with several examples of tagged architectures.

3.2.1 Advantages Of Tagged Architectures -

Code generation is much easier in a tagged architecture because type information is carried in the operands to an instruction, and thus there need not be different instructions which perform the same operation on different types. Certain restricted type conversions can be done automatically (for example, to permit a

floating point and integer to be added), and error conditions raised if a conversion is not appropriate. Errors such as performing a floating add on an integer value are thus eliminated. Tagging also permits vectors to be introduced as a hardware type, with the vector being defined by a base-bounds pair and perhaps additional information. Since Gypsy insists on type compatibility among unstructured types like characters and integers, the ability to tag a word would simplify the task of compiling Gypsy.

3.2.2 Examples Of Tagged Architectures -

One of the first uses of hardware tags was as trap bits. The Rice Computer (R-1) [Rice 68] had two bits per word which could be used in debugging software. The Burroughs B5000 [Lonergan 61] had a single tag bit per word which indicated that special hardware interpretation was needed; the word was interpreted to refer to some object in storage.

Tags which identified the arithmetic type of a word were built into the Telefunken TR-4 [TR440 70], Burroughs B6500/B7500 [Hauck 68], Iliffe's BLM [Iliffe 69], and the Rice Research Computer R-2 [Feustal 72].

Protection keys on blocks of storage are found on the IBM System 360 [Amdahl 64], RCA Spectra 70, and Xerox Sigma family [Xerox 70]. These keys can be considered to be tags on blocks of

memory.

This list of existing tagged architectures is taken from [Feustal 73].

3.3 Assuring The Functionality Of The Hardware

An essential part of a reliable architecture is that the hardware either function as expected or signal an error status. The error status can be indicated by appropriately setting a status word in the currently executing routine as part of the abnormal return from the hardware routine. The routine resumes execution at an instruction offset indicated for that error condition, just as for abnormal exits from software routines.

The scope of damage by hardware faults can be limited by requiring that the hardware adhere to the same addressing conventions that are imposed on the software, as is done in the Plessey 250. That is, the processor cannot create an absolute address, but must use a capability when accessing memory. Thus damage caused by a fault in the processor is limited to the areas defined by the capability registers. By deliberately invalidating the capability registers upon detection of a fault in the processor, all further damage to memory is prevented. The processor can be granted access to a restricted area of memory for execution of a self-check routine.

If the hardware routines also use capabilities, as discussed in the model proposed in the next chapter, invalidating the current routine register effectively forces the processor to halt, since no instructions can be executed. Alternatively, the contents of the capability registers can be replaced with capabilities for a severely restricted instruction set for self-checking.

Much importance is placed on the integrity of capabilities and address formation functions in a capability based architecture. The capabilities should include redundancy beyond simple parity bits; a checksum or other such redundancy check which is verified on every reference to the capability should be part of the memory access functions. Parallel computation of the checksum with the nonredundant parts of the functions is readily accomplished with current hardware technology, and therefore the redundancy need not slow memory references. Computing parity on both the address transfer to memory and on the data transfer between memory locations is important.

Transfers between memory and disk need to be validated. One good way to accomplish this on reads from a disk is to transfer a word from the disk to a temporary location in the disk controller and validate its parity. If it is good, transfer the value to memory, otherwise begin parity error handling in the disk read function. On disk writes, transfer the memory word to the temporary location, then write it to the disk. Include on the disk head the

ability to read the word just written (as is done on many tape drives), and compare what was written with the contents of the temporary location. If the comparison fails, the disk write function begins its write error handling.

CHAPTER 4

ARCHITECTURAL MODEL

This chapter defines and discusses architectural designs which meet the requirements for effectively supporting modern software concepts. The feasibility of these designs and additional functional features which result from the designs are also discussed.

The fundamental concept of the architectural model is the integration of the concepts of capability based (section 3.1) and tagged (section 3.2) architectures, and the definition of a functionality based on this integrated concept which implements typed data objects (section 4.3), defines the format of code objects (section 4.4), and describes a mechanism for the unification of software and hardware routine calls (section 4.5). The model supports the concept of stack memory and its application to recovery and I/O (section 4.6), and the problem of confinement in the proposed model (section 4.7). The scope of applicability of the model is discussed in section 4.8.

4.1 Application To Gypsy

The language Gypsy [Ambler 76] is a programming language which incorporates in a coherent manner modern programming language and software concepts. Analysis of the effectiveness of the proposed architecture in supporting the implementation of Gypsy therefore constitutes a reasonable test of the completeness with which the proposed architecture meets its goals. The next sections briefly define the format in which Gypsy implements its conceptual base, and discuss the properties of capabilities and tagged architectures with respect to the Gypsy implementation, and the means by which the integration of these concepts provides the framework for meeting the goals of the model architecture.

4.1.1 Modular Programming -

Gypsy defines strongly modular program units which can interact only through rigidly defined interfaces. In particular, there are no global variables in Gypsy; a routine has access only to its parameters and local variables. I/O is included in this restriction; files are parameters. Entry and exit specifications are part of Gypsy routines, and may be marked as runtime validated rather than prooftime proven. The entry condition states a predicate on the parameters which must hold on routine entry; the exit condition is the predicate on the parameters which must hold on

routine exit. The routine management features, defined in section 4.5, provide rigorous call-return interfaces at which runtime validatable assertions, such as entry and exit specifications, can be tested. Capabilities (section 3.2) provide a simple but complete means to specify the address space of a routine.

4.1.2 Protection And Sharing -

There is no nesting of routine definitions in Gypsy. Gypsy implements protection of both routine and data objects through the use of access lists associated with the definition of each unit. A unit may define a constant, type, function, procedure, or program. One kind of access list explicitly states which units may reference the unit being defined. For a type, referencing means declaring a parameter or local variable of that type; for routines, referencing means calling that routine. Type definitions may have another access list which explicitly states which units may know the representation of variables of that type; for example, that a given type is a record, and contains certain fields. In this manner, data abstraction can be implemented. Routines granted reference but not representation access can treat variables of that type only as indivisible entities; that is, declare variables and parameters of that type and pass them as parameters. Access lists can be implemented directly with capabilities, by requiring that a routine

which desires to use a type or routine definition possess a capability for that definition which contains sufficient access rights.

4.1.3 Fault Tolerance -

Gypsy has strong type checking. The type mechanism described in section 4.3 directly implements type checking, not only for simple types, but also for user-defined types. This will directly increase runtime efficiency of Gypsy programs and also ease the task of compilation. The unified call-return mechanism (section 4.5) provides a means for isolating and characterizing faults detected at routine boundaries.

4.1.4 Concurrency -

Gypsy permits the concurrent execution of procedures via a cobegin statement. The only call-by-reference parameters which can be shared among concurrently executing procedures are structured objects called buffers, which must be passed as parameters from some common ancestor. Buffers are the sole means of communication among concurrent procedures. A buffer is a bounded first-in-first-out queue of objects of a single but arbitrary type, which is stated in its declaration. Procedures may send objects to the buffer or receive objects from it; an attempt to send to a full buffer or

receive from an empty one results in blocking of the routine until the buffer state changes to permit the operation to complete.

Explicit provisions for routine management in the hardware simplifies the management of concurrency. Additionally, the message concept is directly implementable in terms of capabilities. Therefore, the model architecture directly supports concurrency as it is expressed in Gypsy.

4.2 Overview Of Proposed Model

A storage management design is described which unifies the concept of capabilities and word tags, and also permits uniform naming of objects whether they are software entities like integer vectors, or hardware devices, like disks.

In this design, everything represented in the system is of some type and is an "object". "Objects" include software routines and hardware objects such as random access memory and functional units (such as adders). All objects are referenced solely through descriptors, which provide a non-forgeable access path to an object.

A storage device is an object in which other objects may be placed; descriptors and the operations on them are implemented on storage devices. Virtual memory is one such storage device. Since storage devices are designed to be partitioned into objects, and these objects can be further partitioned, objects contain headers

which describe where their component objects are placed.

A descriptor consists of a type field, access rights field, and an object id field (see figure 2). The type field denotes the type of the object being described, the access rights field denotes which operations are allowed on the object (using this descriptor), and the object id field denotes where the object is. The object id in a descriptor denotes the object immediately containing the object being described, and the address in that object of the described object. The nature of this address varies with the storage device.

It is obviously impractical to include a full descriptor for each object in the system; for example, using six bytes to describe each two byte integer in a thousand word vector. A simple mechanism is proposed to make the descriptor mechanism practical. One of the basic kinds of objects is the "type" object. Instead of storing full type identifiers, access rights fields, and so on, in descriptors, we can store only a minimal amount of information in the descriptors, and let this information denote where the full information is to be found in the appropriate "type" object. The rationale for storing the information in "type" objects is that objects of the same type probably use many of the same fields in their component descriptors.

The primitive operations on descriptors are:

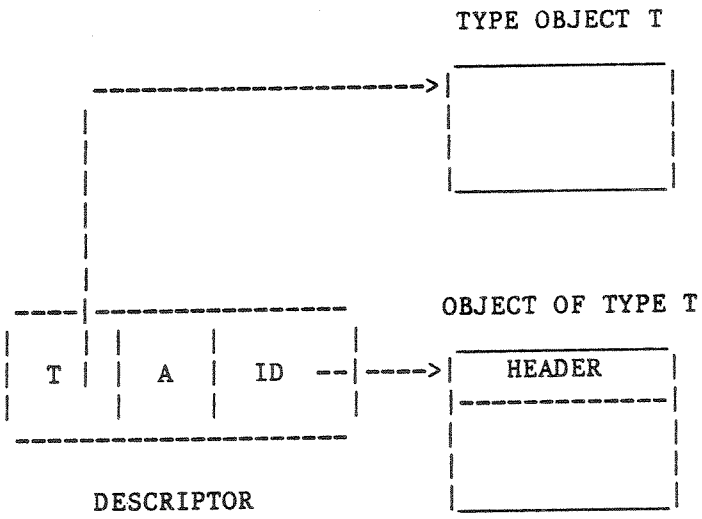


Figure 2. Fields of a descriptor

1. new_object (D, T) : Change the type field of descriptor D to T. The object id field is set to denote an empty object immediately following the descriptor. T is a descriptor for a type object.
2. modify_rights (D, A) : Set the access rights of descriptor D to A.
3. copy_descriptor (S, D) : Copy the access rights field of descriptor S into the corresponding field of descriptor D, and change the object id field of D so that it denotes the same object as does S. The types of D and S must be the same.

Parameters to routines are contained in objects shared between the calling and called routine. The normal type checking mechanism of the copy_descriptor operation enforces the property that descriptors placed in this object are of the correct type and have adequate access rights.

One important class of objects is a "routine" object. A routine object contains objects which define the execution state (such as program counter and status words), an object which defines its function (the code block), and the base descriptor object (consisting of descriptors for objects which are directly addressable by the code). A code object is a vector of instruction objects.

In most contemporary architectures, the mechanisms to call software routines are distinct from those to call hardware routines (i.e., machine instructions). This is counter to the information hiding property inherent in the concepts of modularity and abstract data structures, in that use of the routine thus depends on its implementation. Using the storage management design described in the section 4.3, a design for routine management in which hardware devices are treated identically to the way in which software routines are treated is described in the following sections.

The concepts of software routine, hardware functional unit, and hardware device are unified in this model under the term "device". All devices have three basic components: state, function, and base descriptor. A "controller" is the interpreter of the "function" part of a device, in that it analyzes the function part to determine which other devices are to be called with what parameters in what order. The function part operates on the "base descriptor object" part; that is, the base descriptor object is the data of the device. The "state" is data of the interpreter.

There are two synchronization primitives in the model: "attach" and "release". They are basic to guaranteeing exclusive access to shared objects. "Attach (S, D) is defined as: Attempt to gain exclusive access to shared object S; if successful, return a descriptor for it in D, else do nothing. "Release (S, D)" is defined as: Release exclusive access to shared object S; on

return, D no longer names the shared object.

4.3 Unification Of Capabilities And Tags

In this design, everything represented in the system is of some type and is an "object." "Objects" include software routines and hardware objects such as random access memory and functional units (such as adders). All objects are referenced solely through descriptors, which provide a non-forgeable access path to an object. The routines which know and can manipulate the internal representation of objects of a particular type are collectively called the manager for objects of that type.

Storage device objects (described in the next section) may be built from other objects. A basic storage device built from other objects is the "virtual memory object". The "virtual memory object" builder routines make this object appear to be a very large object in which objects (called virtual memory objects) of other types can be created, so that what would be called files in a conventional architecture are in the address space (like MULTICS).

Another primitive kind of object is "type." Using the type managers and storage devices (like virtual memory), arbitrary types and objects of these types are created and manipulated.

4.3.1 Storage Devices -

A storage device is an object in which other objects may be placed; descriptors and the operations on them are implemented on storage devices. Virtual memory is one such storage device. Since storage devices are designed to be partitioned into objects, and these objects can be further partitioned, objects contain headers which describe where their component objects are placed. The structure of this header varies with the storage device; in virtual memory, for example, the header denotes the displacement of the components (see section 4.3.3), whereas in an associative device, such as a self managing secondary memory [DeMartinis 76], the header may contain the labels of the components.

A distinction is made between a device builder for a storage device, and the manager of the storage device object. The device builder implements descriptors and header operations on the concrete realization of the device, such as random access memory and disk tracks; the manager of the storage device object treats the storage device as an object which can be partitioned into objects, using the descriptor and header operations.

4.3.2 Descriptors -

A descriptor consists of a type field, access rights field, and an object id field (see figure 2). The type field denotes the type of the object being described, the access rights field denotes which operations are allowed on the object (using this descriptor), and the object id field denotes where the object is.

It is obviously impractical to include a full descriptor for each object in the system; for example, using six bytes to describe each two byte integer in a thousand word vector. A simple mechanism is proposed to make the descriptor mechanism practical. Recall that one of the basic kinds of objects is the "type" object. Instead of storing full type identifiers, access rights fields, and so on, in descriptors, store only a minimal amount of information in the descriptors, and let this information denote where the full information is to be found in the appropriate "type" object.

The rationale for storing the information in "type" objects is that objects of the same type probably use many of the same fields in their component descriptors. A "type" object is composed of three objects for each of the three fields of a descriptor and three objects for each of the fields of a header. The three objects which describe a given field of a descriptor or header are the two possible sizes of that field (say, in terms of bits) in a descriptor or header, and an object which is a vector of objects of that kind of field. The fields of a descriptor are type, access rights, and

object id. For virtual memory objects, the header fields are element size vector, displacement vector, and correction vector (described in section 4.3.3.1).

The minimum addressable unit in this model is deliberately left unspecified. The unit chosen in a specific implementation of the model is not important in the definition of the model. The header of an object describes where each descriptor contained in that object begins; the intent of the descriptor and header formats is that in a scan beginning at the start of the descriptor, the descriptor interpretation function can determine which how long each descriptor field is.

The format of a descriptor is as follows:

```
record case all_default : boolean of
    true: ({use default type, access rights, object id});
    false:
        (Type:
            case what_type: (Default_type, Type_Index,
                    Explicit_Type) of
            Default_type:
                    ({use default value in type object});
            Type_Index: (ITV: Index_into_type_vector);
            Explicit_Type: (TI: Type_id)
           end;
        Access_Rights:
            case what_rights: (Default_rights,Rights_Index,
                    Explicit_rights) of
            Default_rights:
                    ({use default value in type vector});
            Rights_Index:
                    (IAR: Index_into_access_rights_vector);
            Explicit_rights: (ARV: Access_rights_vector)
            end;
        Object_id:
            case what_id:(Immediate,Id_Index,Explicit_Id) of
```

```
                Immediate: ({object is here});
                Id_Index: (IOIV: Index_into_object_id_vector);
                Explicit_Id: (OI: Object_id)
                end
            )
        end;
```

Specific operations are provided by the storage device builder to operate on descriptors; even hardware functional units are constrained to use these operations. The interpretation of a descriptor proceeds as follows (see figure 3 for a description of the interpretation of two descriptors for an object of type T2).

The type field is interpreted using the type object (here called T1) of the object containing the descriptor being interpreted. If the what_type field is Default_type, the default type is found at a standard offset in the vector of type objects in T1. If the what_type field is Type_index, the "size of type index" object in T1 is used to determine the length of an index. This index specifies the desired "type" object in the type object vector in T1. If what_type is Explicit_type, the "size of type object" field in T1 is used to determine the length of the type object (really a descriptor for a type object), which follows immediately in the descriptor.

Then, using the "type" object specified by the Type field (here called T2), the access rights and object id are interpreted. The access rights field is interpreted in the same way as the type field, except that an index into the vector of access rights objects

in T2 may be found. The object id field is interpreted somewhat differently. If what_id is Immediate, the object directly follows the descriptor. When used in this way, the descriptor looks like a tag in a tagged architecture. If what_id is Explicit_id or id_index, the location of the object being described by the descriptor is expressed as either an object id or an index into the object id vector of T2 (in the same manner as the access rights). Figure 3 gives an example of descriptor interpretation.

4.3.2.1 Operations On Descriptors -

Since types play such an important role in defining and using abstract objects, the hardware should make provisions for indicating and checking the types of software objects. A somewhat awkward technique for doing this solely with untyped capabilities is described in Appendix B of [Good 77]. However, the model being described directly provides this facility. The primitive operations on descriptors are:

1. new_object (D, T)
    1. Operation: Change the type field of descriptor D to T. The object id field is set to denote an empty object (which takes no space) immediately following the descriptor. T is a descriptor for a type object.

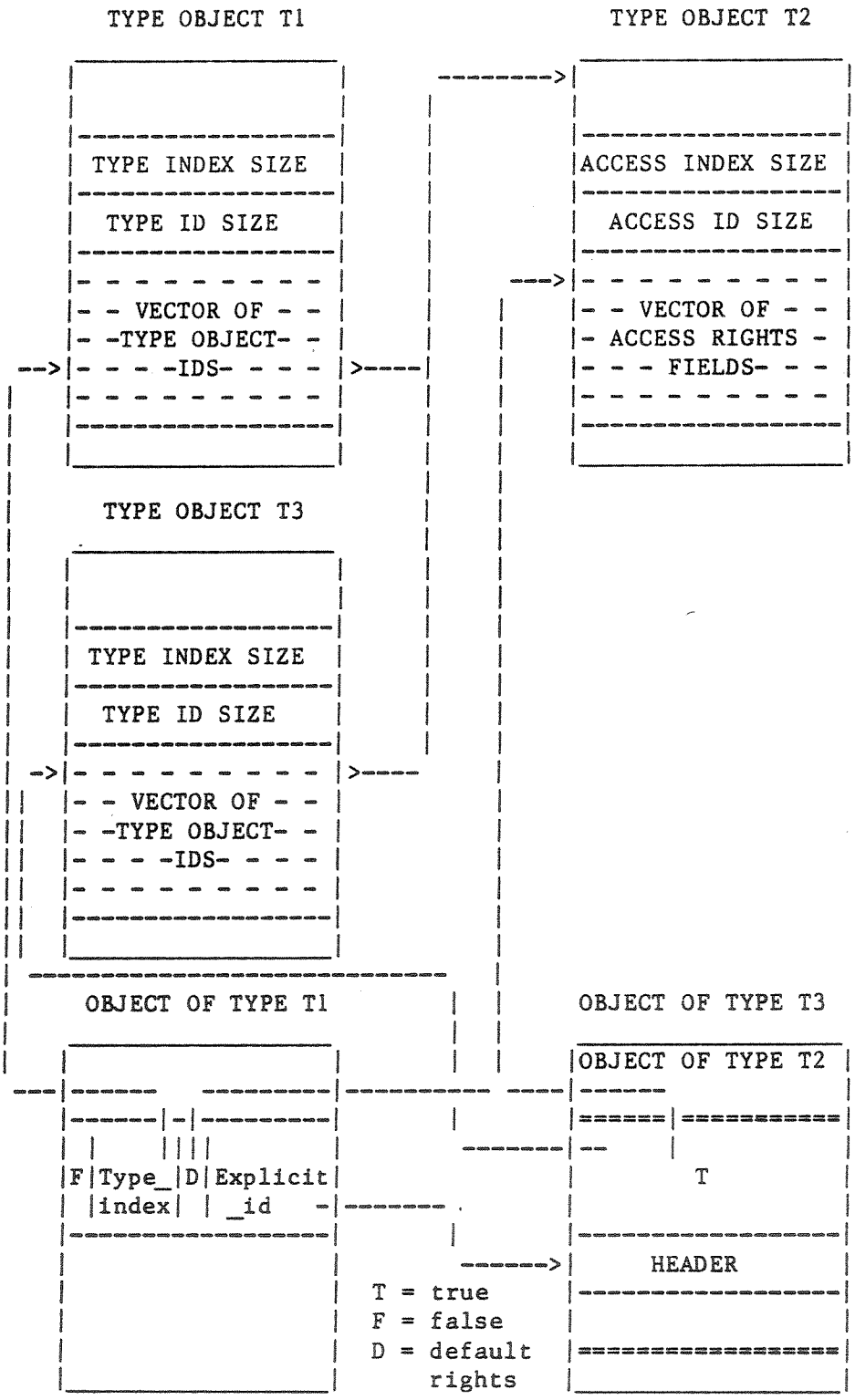


Figure 3. Two descriptors for one object of type T2

2. Comments: The access rights of the object originally named by D must permit the type and object id fields to be changed.

2. modify_rights (D, A)

   1. Operation: Set the access rights of descriptor D to A.
   2. Comments: This operation allows either an increase or decrease of access rights for any type. A system provided (but not primitive) operation can be used to generate a routine which applies the modify_rights operation only to descriptors of specific type or types. These routines should only be callable by those routines which can access the internal representation of an abstract object (i.e., operations on the type). Other routines may call something like "reduce_rights", which in turn calls modify_rights.

3. copy_descriptor (S, D)

   1. Operation: Copy the access rights field of descriptor S into the corresponding field of descriptor D, and change the object id field of D so that it denotes the same object as does S.

2. Comments: The types of S and D must be the same. S must contain the access right to be copied, and the old access rights of D must permit changing the object id field.

If the object immediately follows the descriptor, the access rights on the descriptor probably prohibit overwriting the type and object id fields. For example, if another descriptor's object id refers to a given object, and the descriptor immediately before that object changes its type field, the two descriptors denote different types for the same object. This inconsistency would be undesirable.

To create a new object, a routine calls the type manager for the type of object being created. Parameters to that manager specify a containing object and the address in that object at which the descriptor and the object are to be created; that is, this is a structure modification operation. If the containing object is a storage device object, then this is effectively a create operation. The appropriate access rights to the existing object are required, and various limitations, such as size limitations of the element, are enforced by the header operations which are used to effect the object creation. The type manager invokes the storage device object builder to obtain an object of that type in the specified object. The type manager may in turn create descriptors (to existing

objects) and/or descriptor-object pairs (to new objects) in the new object. This nesting of creation calls stops when primitive objects of the storage device are created, which are initialized by the device builder of the storage device.

Parameters to routines are contained in objects shared between the calling and called routine. The normal type checking mechanism of the copy_descriptor operation enforces the property that descriptors placed in this object are of the correct type and access rights.

4.3.2.2 Object Id -

Up to this point, little has been said as to exactly what an object id is. Basically, the object id in a descriptor denotes the object immediately containing the object being described, and the address in that object of the described object. The nature of this address varies with the storage device. The address is really a parameterized call to the accessing function for the storage device, with the name of the function being implied by the kind of storage device. The address could be anything, such as a disk seek function and track number, or an associative function and unique name.

4.3.3 Virtual Memory Objects -

The basic unstructured object in virtual memory is a bit vector, preceded by a header which gives its length. Primitive operations on bit vectors, such as shift and boolean functions, are provided by the virtual memory device builder. The actual format of the header is not important; what is important is that it provide a concise representation of where the components of an object start, which can be interpreted in a reasonable amount of time. A specific example of such a representation follows.

All nonprimitive objects in virtual memory consist of a header, and F "fields", each of which is a group of consecutive elements of size S[i], where i is the field number. An element is a descriptor and possibly an object. The header consists of three vectors: an index vector I, which gives the index in the object at which each field starts, a size vector S which gives the size of each element in a given field, and a correction vector C which facilitates rapid computation of the location of a particular element in the object, where (C[i] = (SUM ((I[j+1] - I[j]) * S[j]), j=1,i-1) - I[i]*S[i], i=2,F). C[1] = I[1] = 0 and need not be stored. The summation is simply the total size of all fields preceeding the one desired. Fields are numbered 1 to F, and indices start at zero. There is an additional entry at the end of I which has a value one greater than the largest index of the object. See figure 4 for an example of a header and an object.

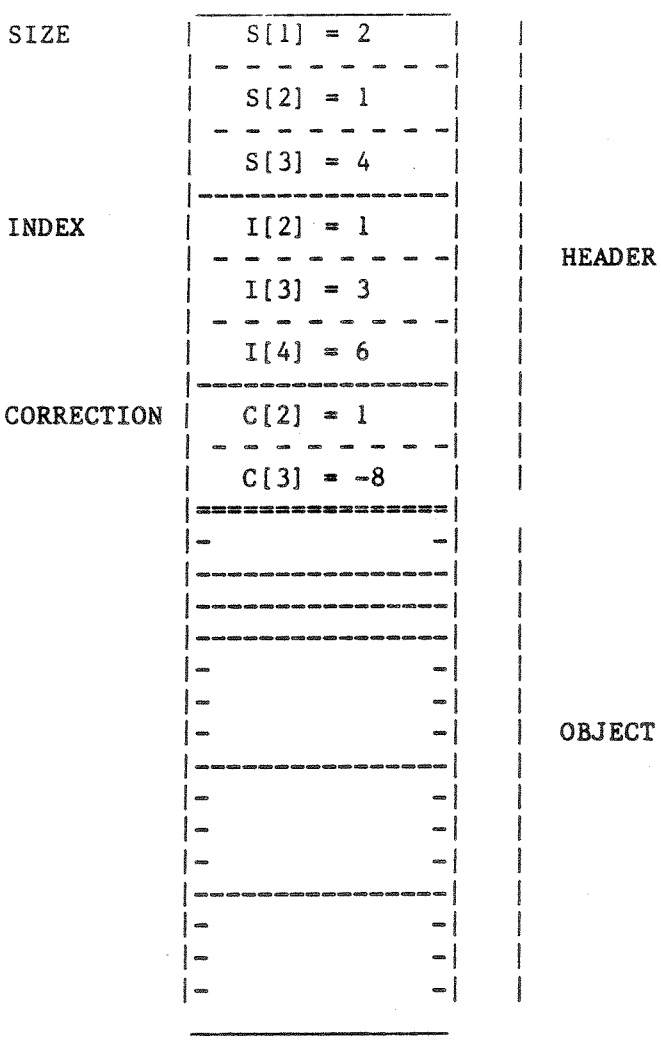


Figure 4. Object Header

Elements of an object are accessed by an index into the object. The offset of the desired element from the start of the object can be found in two steps. The algorithm is:

```
begin
cobegin
    B[i] := Index ≥ I[i], i=1,F+1;
    A[i] := Index * S[i], i=1,F;
end;
using i in [1..F] such that B[i] and not B[i+1],
    offset := C[i] + A[i]
else signal index_error
end.
```

Notice that an index_error condition is detected by the predicate "B[i] and not B[i+1]" failing for all i in the range 1..F.

For a homogeneous array F is 1, and I[2] is the number of elements in the array. The access algorithm reduces to:

```
begin
cobegin
    B[1] := Index ≥ 0;
    B[2] := Index ≥ I[2];
    A[1] := Index * S[1];
end;
If B[1] and not B[2]
then offset := A[1]
else signal index_error
end
```

For Gypsy or Pascal records, in which each field is of a different type, I[i] = i - 1, i = 1, F+1, and C[i] = (SUM S[j], j = 1, i-1) - (i - 1) * S[i], i = 2, F. The algorithm reduces to:

```
If (Index ≥ 0) and not (Index ≥ F)
Then offset := C[Index+1] + Index * S[Index+1]
else signal index_error
End
```

Notice that offset = (SUM S[j], j = 1, Index).

4.3.3.1 Virtual Object Headers -

The header of a virtual memory object is represented as:

```
record case all_default : boolean of
    true: ({use default size, index, correction vectors});
    false: (Size, Index, Correction:
            case what : (Default, Index, Explicit) of
            Default: ({use default index});
            Index: (I: Index_value);
            Explicit: (SIC: size_index_or_correction_vector)
           )
    end;
```

The interpretation of the Size, Index, and Correction vectors is the same as for the access rights field of the descriptor used to access the object (e.g., the Size field is an index into the vector of size vectors in the type object). One anticipates that the element size fields would most often be a power of 2, so that the multiplies are replaced by shifts. Note that the user of an object need not know the internal layout of the object it is accessing; that is, whether the elements are stored immediately after the descriptor, or if in another object completely, where the item is physically located in that object.

The primitive operation on virtual object headers is modify_header (D, H), defined as: Modify the header of the object named by D so that it is described by H. If the number of elements is increased, the new elements are of the null type (whose objects are all of length zero and contain zero elements). If the number of elements is decreased from j to i, elements i+1 to j are destroyed. References to elements which existed before the call to this operation, and were not destroyed, are not affected.

The right to perform this operation must be contained in the access rights of D. If objects are destroyed, they all must have access rights which permit destruction.

Objects are named by their index in another object ("the nth element of object O"), not their physical displacement in that object. This operation maintains this independence of physical displacement by relocating elements as necessary when element sizes are changed, so that the same index will refer to the same object. If D denotes "virtual memory", then this instruction acts like a create or destroy operation. This operation enforces the property that an object named by one descriptor overlaps with the object named by another descriptor only if either the same object is named, or if one is a component of the other.

Since a descriptor always immediately precedes an object, consistency checks can be made between any distant descriptor used to reference the object and the descriptor in front of the object.

For example, the types can be compared.

To delete an object without changing the header of its containing object, its type is changed to the null type. The name of the deleted object still exists (as an index in its containing object), but any reference to it will result in an error, due to type mismatch. Any reference to an object contained in the deleted object will fail, since objects of the null type have no elements. The series of indices used to name a virtual memory object can thus be used as a system-wide, non-reused unique id, at the cost of monotonically nondecreasing indices.

4.3.3.2 Object Ids For Virtual Memory Objects -

When an object immediately follows its descriptor, no object id need be stored, but if another object is to be given a descriptor to this object, an object id of the form [object id of virtual memory, index 1, ..., index n] is stored either in the new descriptor or in the "type" object of the shared object, at the time that the new descriptor is created. The first term names the storage device (virtual memory). The index 1 term specifies an index into virtual memory, index 2 an index into the object denoted by index 1, and so on, until the desired object is denoted. This name is nothing more than a recursive application of the convention stated in the first paragraph of this section; [object id of

virtual memory, index 1, ... index n-1] is the containing object, and "index n" is the index in the containing object of the desired object. The depth of nesting would in most cases be less than 3 or 4.

All "type" objects are themselves of the same type, and therefore "type" object ids are all indices into the same master "type" object. This object is directly contained in virtual memory and thus the location of "type" objects is known.

4.3.3.3 Internal Storage Management Structure -

How the virtual memory device builder implements virtual memory is unimportant to the conceptual design of this system; however, a paged structure is described as an example of how the storage management system can relate to physical virtual memory. Using conventional terms, the virtual memory object builder maintains a very large object (called virtual memory). The physical displacements of objects can be split into page number and offset fields. The virtual memory device builder maintains a table which contains descriptors for page frames in physical (not necessarily random access) memory. The location of the pages may be on disk, for example. To implement virtual memory, storage devices called "random access memory" and "disk" are used, along with routines which manage these devices. These devices need not be known to

users of the virtual memory. In order to read (or write) an object (say, a page) from (or to) a disk, the descriptor for the location on disk of the desired object is given to (or obtained from) the disk I/O manager, along with a descriptor for where the object is to (or does) reside in random access memory.

4.4 Routine Objects And Instruction Formats

One important type of object is "routine". A routine object contains objects which define the execution state (such as program counter and status words), an object which defines its function (the code block), and the base descriptor object (consisting of descriptors for objects which are directly addressable by the code). A code object is basically a vector of instruction objects, each of which consists of one or more "descriptor path" objects. Each of these descriptor path objects denotes a descriptor by specifying a path to the descriptor, using whatever naming convention is appropriate to the particular object in the path; for virtual memory objects, for example, these names are indices into objects. The first name is an index of an object in the routine's base descriptor object. The second name denotes an object in the object named by the first name, and so on. Like all other objects, descriptor path objects have a header, and this header denotes how many elements (i.e., names) are in the descriptor path object.

The first descriptor path object in an instruction denotes an input queue for the routine to be invoked, and the others are parameters to be placed on that queue. The header of the instruction object denotes how many parameters are in the instruction (i.e., the number of components in the object, as for all objects). Since descriptors contain the type of the object, validation that formal and actual parameters are of the same type is directly provided.

This structure of code blocks eases the burden of the compiler in that problems concerning variable length instructions are lessened, such as placing the instruction on certain word boundaries. At runtime, the processor knows where instruction boundaries are located without interpreting the instructions, expediting prefetching and other such high speed techniques.

The question may be asked: "Where do machine instructions fit into this proposal?" One of the descriptors in the base descriptor object names an object which contains descriptors for the queue for machine instructions routines. This object may really be in a control store, but this fact is irrelevant to the code. Since the object containing the machine instruction descriptors can potentially be changed by routines having appropriate access rights for it, the way is open for very flexible order codes. Thus, different routines may use different instruction sets, by virtue of having different descriptors for the machine routines. A manager of

machine instruction objects can be built to permit users to design their own microcoded instructions, with the manager guaranteeing that the generated instructions are of such form that they cannot adversely effect system integrity.

4.5 Unification Of Software And Hardware Routines

In most contemporary architectures, the mechanisms to call software routines are distinct from those to call hardware routines (i.e., machine instructions). This is counter to the principle of information hiding inherent in the concepts of modularity and data abstraction, in that use of the routine thus depends on its implementation. In a capability system, software routines are called by specifying an offset to the start of a block of capabilities. The capability thus denoted names the routine to be called. The parameter passing mechanism makes capabilities for the parameters available to the called routine. On the other hand, calls to hardware routines consist of parts of or several words containing an operation code, index register fields, accumulator register fields, address offsets, and various modifier bits. Using the storage management design described in the preceding section, a design for routine management in which hardware devices are treated identically to the way in which software routines are treated is described in the following sections.

The concepts of software routine and hardware functional unit are unified under the term "device". All devices have three basic components: state, function, and base descriptor. A "controller" is the interpreter of the "function" part of a device, in that it analyzes the function part to determine which other devices are to be called with what parameters in what order. The function part operates on the "base descriptor object" part; that is, the base descriptor object is the data of the device.

The "state" is data of the interpreter. The controller can invoke operations on the components of state objects; i.e, it is the manager of objects whose type is "state". It thus need not know the internal structure of the components of state objects. The state may consist of different objects, and have different operations, for different devices. For example, the state of a disk device includes the current position of the disk and heads, and whether the disk is performing an operation like seek or read; electromechanical and electromagnetic hardware perform the operations on the disk surfaces and return information in objects (usually called incore buffers) shared with callers of these operations.

Software routines are also devices; their state information includes current instruction location, whether running or waiting, etc. These software devices can be created dynamically, whereas hardware devices are static, physical entities; both are treated

identically. The routine object is described in section 4.4.

Part of the base descriptor object of a device is shared data objects which may be organized as FIFO queues, stacks, etc. For ease of reference, these objects will be called queues, regardless of the discipline by which entries are inserted and removed. Devices may cause themselves to become suspended (using appropriate calls to its controller) while waiting for information, such as requests for service, to be placed on these queues. As will be described later, these queues are special objects only in the sense that some other device polls them and takes some appropriate action when something is placed into them. Devices are treated like processes in an infinite loop.

4.5.1 Synchronization Primitives -

There are two synchronization primitives in the model: "attach" and "release". They are basic to guaranteeing exclusive access to shared objects.

"Attach (S, D) is defined as: Attempt to gain exclusive access to shared object S; if successful, return a descriptor for it in D, else do nothing. If the object is not available, the routine calling attach can ask the routine's controller to place it on a wait queue associated with the object, but this scheduling is not part of the attach operation itself.

"Release (S, D)" is defined as: Release exclusive access to shared object S; on return, D no longer names the shared object. The routine calling release may request that a routine on the wait queue associated with the object be reactivated, but, again, this action is not part of the release operation.

Synchronization primitives for processes can be implemented with the support of the capability system and simple hardware. In particular, indivisible test-and-branch and increment instructions can be used to implement attach and release. If two devices are to participate in a synchronized sharing of data, they receive as parameters the same synchronization object (called a buffer in Gypsy). This object includes a queue of data objects sent and not received, a queue for devices which are blocked awaiting data on an empty data queue, and a queue for devices blocked trying to send data to a full data queue. More complex synchronization objects, such as monitors, can also be built from the primitive operations.

4.5.2 Calling A Device -

A device call proceeds as follows. Some device places data on an input queue of the called device. The controller for the called device senses data on the input queue and activates the device. A descriptor for a queue shared with its controller is available to the device; it can thus use this descriptor in order

to communicate with its controller. For software devices, this controller might be called a scheduler or policy module. For hardware instructions, the controller is usually called the instruction fetch / decode unit. For I/O devices, the controller is usually called an I/O controller. Notice that the call never actually refers to the called device; it only refers to queues. It is entirely possible that several devices could service the same input queue.

Notice that since controllers are themselves devices, they can receive input on queues, and use this information in affecting the state of devices which they are controlling. For example, a controller might receive a halt request from either the controlled device, or some other device which is allowed to perform that operation (that is, possesses appropriate descriptors).

How this call mechanism works is described for a common construct in programs: the procedure call. The desired effect is: A device wishes to call another device, and suspend itself while the called device is executing; when the called device has performed the desired action, it is to terminate and cause the calling device to continue. An implementation of procedure calls is:

1. The called device is normally suspended, waiting for data in a queue on which parameters are to be placed.

2. The calling device places the parameters on the queue, and becomes suspended by requesting its controller to reactivate it when something appears on a completion queue shared between the caller and called routine.
3. The controller for the called device senses data on the queue, and activates the called device.
4. When the called device wishes to return, it places a completion code on a completion queue, and becomes suspended waiting for data in its parameter queue.
5. The caller's controller senses input on the completion object, and either continues the caller from the place of the call, or forces a branch (by modifying the state) to an error condition handler in the calling device, depending on the value of the completion object. The same treatment of return condition codes applies whether the condition denotes arithmetic overflow in an adder, memory parity fault in a memory access, or table overflow in a compiler symbol table.

It is often the case that the called device is created by its controller for one invocation, and is destroyed when it returns.

4.5.3 Software Devices -

A hardware device can service its own queues and perform its function directly, but software devices are defined in terms of calls to other devices. Devices called CPU dispatchers are utilized to help interpret software devices (i.e., as servers). The input queue to all the CPU dispatchers is named the system ready queue, and the ready queue manager has sole input access to it. (This concept of a ready queue is derived from the process ready queue in the Plessey 250, section 2.3.1.)

The controller for a software device calls the ready queue manager in order to place a descriptor for a software device on the ready queue. A CPU dispatcher unit takes the top entry (descriptor) off the ready queue and analyzes the current instruction in that device, finding the descriptors for the queue and the parameters. The parameters are placed on that queue using a FCFS (or other predetermined and fixed) discipline. A CPU dispatcher is thus much like the instruction fetch / decode unit of an conventional CPU.

For devices with software controllers, "call a device" implies that provision must be made to activate the controller for the device; unlike hardware devices, software controllers cannot continuously poll a queue in order to sense data on it and activate the device. Software devices only execute insofar as the CPU dispatcher processes their instructions. This activating of the controller can be done either by running the controller at fixed

time intervals (so that it can poll the appropriate queues), or by having the controller suspend itself on a queue managed by a hardware device; the software controller is awakened via a call from that hardware device (for example, I/O interrupts). In order for instruction execution to proceed at a practical rate, the code and ready queue must both reside on fast memories directly accessible by the CPU dispatchers and ready queue manager.

4.5.4 Registers -

There are no user-accessible processor registers. The availability of high speed but moderately priced random access memories is lessening the advantage of having high speed users-accessible processor registers; the Texas Instruments 990 [Texas 76] system includes the feature of designating an arbitrary part of memory to be used like registers. The overhead of maintaining registers (which amount to another level of memory hierarchy) can easily exceed any speed advantage. In the model being described in this dissertation, the only context that must be switched when a processor changes from one routine to another involves changing the descriptor which names the currently executing routine object (and nothing at all if interpretation of code is done instruction by instruction from the ready queue). Context switches are therefore very fast.

Besides speed, registers are also used in conventional architectures because they can be named in an instruction using only a few bits. The base descriptor object provides this feature, since indices into this object can be in the same range as one might find for register names. Notice that if a descriptor in this object states that the object immediately follows the descriptor, such as an integer, that element in the base descriptor object acts like a data register; if it denotes a large, complex object located elsewhere, the element acts like a capability register in a capability based machine.

4.5.5 Various Aspects Of Device Management -

A device called the real time clock driver maintains a data object called the real time clock. Other devices, especially controllers, may consult the clock in order to do their scheduling.

The relationship between a controller and a controlled device is very flexible, being determined by which device currently holds a descriptor for the state object of another device. It is quite possible that one device may wish to pass the role of controller to another device. An example of this is found in Hydra [Levin 75], where a high level scheduler (policy module) relinquishes short term scheduling to another scheduler, in the kernel, which in turn delegates execution of machine instruction

sequences to the CPU.

An external interrupt is simply a request that a given routine be given high priority for CPU service. Trivial interrupt service routines, such as accepting a character from a slow serial device, are better handled by hardware devices, which place the data on queues which can be serviced in larger bursts by software routines (as is done in the Plessey System 250, section 2.3.1). A trap, or interrupt internal to a processor, becomes no more than an error return from a hardware device. Any device can call any other device (given the appropriate descriptor), and the fact that the caller may be hardware and the called device software is irrelevant.

Notice that the scheduler need not be able to access the local data of the routine being controlled. This is an essential property for verification. However, for debugging purposes, a user defined scheduler can be provided which allows a calling routine to act as a debugging supervisor (install breakpoints, read and modify local data, force control, etc.).

The operating system starts as a single routine which creates parallel execution streams, each of which can also create parallel execution streams, to an arbitrary depth. Some of these streams are called user jobs, running under a system job manager process. Others are system tasks such as the disk manager. Users may create and control several parallel execution streams. The designation given to a particular stream is up to the system

designer.

4.6 Stack Memory

One technique which has been suggested for recovery in a modular environment is use of a recursive cache [Randell 75]. The basic unit of recovery in this technique is the recovery block, in which several alternatives are provided for a given computation, similar to the hardware concept of sparing. Essential to the execution of the recovery block is the ability to restore the variables in a recovery block to their values on entry to the block, upon detection of an error in the block. The recursive cache is used to store the value of a variable when the first change is made to it in a recovery block; restoration of the entry value of a variable is done using the values stored in the cache. Upon successful termination of a recovery block, the values stored for that block are discarded from the cache. Recovery blocks may be nested, thus the "recursive" label. Finding an efficient implementation is one the main problems with this approach to recovery. The concept of a stack memory as a solution to this problem will be discussed later in this section.

Input/Output processing software tends to be bulky and inelegant, full of timing constraints, interrupt handling, different formatting and command conventions for each device, and other

problems. The concept of stack memory also deals with this problem.

The basic concept of stack memory is that each memory address is seen by the software as a stack with infinite depth. In terms of implementation, a conventional memory is treated like a hash table, using addresses as keys. A memory processor provides functions which make the table appear to be a stack. The microinstructions of the processor are read top of stack, replace top of stack, and pop a value off the stack. A parameter to the processor determines the actual maximum depth of the stack; excess values are dumped to secondary storage. The secondary device for a given address can be defined, and thus this word acts like a device data register, without the need for an interrupt for each word transferred. Other addresses may hold values denoting, for example, what data is to be read from the device (like a disk address). If the stack is full and a push is done on an address, the routine executing the instruction is delayed in much the same way as with virtual memories when a page must be brought into memory. In terms used in the section on processors, a trap to a scheduler is forced in the central processor in order to give the processor to some other routine.

If a word is used as an I/O data register, a system much like the hardware queues of the Plessey System 250, described in Section 2.3, is obtained by directly using the memory processor's instructions. One word used in this way can be designated for the

event trace (Section 2.4.2).

An implementation of the recursive cache is also straightforward. In the following instruction descriptions:

1. "mod_stack" is a word associated with a routine, used for holding the addresses of the words which have been modified in the current recovery block.
2. "level" is a word associated with a routine which gives the current depth of nesting of recovery blocks.
3. "level_val" is a macro which expands to "read (level, 0)".
4. "marker(i)" returns a distinguished value different for each value of i. Its type can be used to distinguish marker values from user definable values.
5. "push (i, value)" means push value onto stack i.
6. "replace (i, n, value)" changes the contents at depth n of stack i to value. Depth 0 is the top of the stack.

The macroinstructions which follow are described in a Gypsy-like syntax. In particular, "if" statements have the form "if <boolean expression> then <statement list> else <statement list> end" or "if <boolean expression> then <statement list> end". The "leave" operation exits the innermost loop which contains the "leave".

1. read (i, n): word_value; returns the value n levels deep in stack i.

2. enter_recovery_block = begin replace (level, 0, level_val + 1); push (mod_stack, marker (level_val)) end.

3. store (i, value) = begin if read (i, 1) = marker (level_val) then replace (i, 0, value) else push (i, marker (level_val)); push (i, value); push (mod_stack, i) end end.

4. exit_recovery_block (success: boolean) = begin loop i := read (mod_stack, 0); if i = marker (level_val)) then leave end; if success then replace (i, 2, read (i, 0)) end {if}; pop (i); pop(i); pop (mod_stack) end {loop}; replace (level, 0, level_val - 1); pop (mod_stack) end.

4.7 The Confinement Problem And The Architectural Model

A rigorous test of a given set of mechanisms for protection and sharing is the degree to which the classical problems of mutually suspicious subsystems [Schroeder 72], confinement, selective confinement, [Jones 75] and revocation [Redell 74] can be solved. This model architecture includes strong mechanisms for protection and sharing of objects. This section shows how the model architecture solves these problems.

In a computing system in which users agree to share software or data for specific purposes, the problem comes up as to how to prevent transmission of information contrary to the agreed-upon purposes. This is known as the confinement problem. In a system with freely copiable capabilities, there is the particular problem of a dishonest software package passing capabilities provided by another user to a spy routine. Since possession of a capability is considered the right to use it, the spy can now access the stolen object whenever it wishes. The ideas in this section are directly derived from Hydra [Cohen 75].

If a subroutine needs no permanent data, and uses no data abstractions (that is, hidden representations), the owner of the subroutine provides the customer with an EXecute capability for the code. The owner then knows that the customer cannot examine the code, and, since the code block executes only with capabilities provided by the customer and capabilities for new store blocks obtained from the store manager, the customer knows that the subroutine cannot pass information to a destination unknown to the customer.

If the item to be provided is an abstract data object, the descriptor for the object protects the object from examination by the customer, but preventing the operations on the object from passing information to destinations unknown to the customer is more complex than in the preceding case. The ENVironment, NOTPARAMeter,

and MoDiFY access rights are proposed to help solve this problem.

The ENV right is used to control the propagation of descriptors. Without this right, a descriptor cannot be copied into an object which is a parameter. For an arbitrary descriptor C, define the predicates ENV(C) and NOTPARAM(C) such that:

1. NOTPARAM(C) if and only if all descriptors in the descriptor sequence used to access C (starting at the base descriptor object) have the NOTPARAM right.
2. ENV(C) if and only if all descriptors in the descriptor sequence used to access C (starting at the base descriptor object) have the ENV right.

The copy_descriptor operation described in section 4.3.2.1 is extended to enforce the following restriction on the operation "copy_descriptor (S, D)": not ENV(S) implies (not ENV(D) and NOTPARAM(D)).

The modify_rights operation is extended to enforce the following predicates on the call "modify_rights (S, A)":

1. not ENV(S) implies ENV is not in A
2. not NOTPARAM(S) implies NOTPARAM is not in A

To prevent a routine from giving an object passed as a parameter to a component of an abstract object which is shared with some unknown routine, the caller turns off the NOTPARAM and ENV rights in the

descriptors of the objects to be passed as parameters to the suspected operation. The called routine thus cannot copy a descriptor for a parameter, or a component of a parameter, into another parameter. Also, the routine cannot copy such a descriptor into a freshly allocated object and then copy a descriptor for that object into a parameter. However, such descriptors can be copied into a new object which is used for parameters for further routine calls.

The MDFY right controls modification of descriptors. If this right is lacking, the "write" rights of this descriptor are deactivated, and any descriptors accessed via this descriptor lack this right. If this right is turned off in an object passed as a parameter, the called procedure is prevented from transmitting information to a spy via that parameter, since to do this it would have to store something in a block which it shared with a spy, as part of the object. It cannot store information for a spy in a new block for later transmission, since it cannot modify the object to contain the new block.

If a customer is given a descriptor for an abstract object generator, the customer can turn off MDFY, NOTPARAM, and ENV rights when it calls the generator, and be assured that the object returned is composed of completely new blocks and nobody else has a descriptor for it. Thus, it is safe to use that object from then on without restricting MDFY and ENV rights.

Since I/O in the model is done via explicit parameters, for which there is an abstract object generator like there is for any other object, the caller can restrict the called procedure to use I/O procedures of the callers choosing. The caller can prevent the called procedure from communicating with a spy via I/O interference and related techniques [Lampson 73].

4.8 Scope Of Applicability Of Model

The model is designed to be very general, both in terms of its utility in describing existing machines, and in describing new architectures. Many different existing forms of capabilities can be described in terms of the model's descriptors.

For example, a Plessey 250 capability contains a six-bit access rights field and an index into the system capability table. The entries in that table contain, among other information, base and bounds values for a segment. In terms of the model, the type field is degenerate (all objects are of the same "type"). The access rights field of the Plessey capability maps directly onto the model's access rights field. The index into the system capability table maps to the model's object id field. The storage manager routines which are given the capabilities enabling it to treat capabilities as data, and the hardware capability interpretation routines, together form the builder for the Plessey 250 version of

the model's virtual memory object. The capability structure in the Plessey 250 does not extend below the segment level, and thus segments correspond to the primitive unstructured type in the model. All of the hardware data operations must be treated as taking segment objects as parameters (although they may modify only a few words). However, hardware operations are constrained not to access in any way segments for which they are not given a capability.

The hardware machine operations of the model are not specified, and thus any realization of the model is not constrained as to how to implement a given operation. The need on most existing machines to differentiate between software and hardware routines can be viewed as a requirement to explicitly write part of the controller for the software routine, using hardware instructions such as "jump to subroutine".

By considering CPU register set to be an object shared between all tasks running on that CPU, the problems of context switching, both for routine calls and interrupts, can be expressed in terms of the model.

CHAPTER 5

IMPLEMENTATION OF THE ARCHITECTURAL MODEL

In this chapter, an implementation of Gypsy onto physical hardware is described. Gypsy is compiled into the Capability Machine Language (CML), which is the assembly language for the Capability Machine. The Capability Machine is based on the architectural model described in chapter 4. An important difference in the terminology used in the current section and that used in the section describing the model is that unless otherwise stated, "capability" in the current section means the same as "descriptor" used in the model description. The Capability Machine is in turn implemented on a PDP-11/03, which has been supplemented by a simple operating system called the "PROM Operating System". This step is done by translating the Capability Machine Language into PDP-11 assembly language, and then using a standard PDP-11 assembly language assembler. This section first describes CML, and then details of the Capability Machine. Then the design issues relevant to the model and how they relate to the features found in the Capability Machine is discussed. Finally, the implementation of the

Capability Machine on a PDP-11/03 is described.

5.1 Implementation Of The Capability Machine On The Model

This section describes the Capability Machine and its assembly language. The assembly language is first described, and then the machine. The unit definition table and capabilities are discussed. Object allocation and the structure of code bodies follow. The interpreter which corresponds to the CPU, is then discussed. Operations on capabilities are defined. The shared object, which is the sole means of sharing between routine activations, is discussed. Then follows a discussion of how routine calls work, both sequential and concurrent. Scheduling of concurrent routine is described.

5.1.1 The Capability Machine Language (CML) -

Capability Machine Language (CML) is the assembly language for the Capability Machine. Its operations are similar to Gypsy. In both Gypsy and CML:

1. there are similar control structures (compound statement, case, loop), and

2. the same component access operations are provided (subscript and dot).

CML is different from Gypsy in the following respects:

1. The form is one operation per line, rather than nested functions.
2. All variables are explicitly allocated, including temporaries for function evaluation (including operations such as arithmetic).
3. There is no distinction between procedures and functions; functions have precisely one reference parameter which is the result value.
4. Dynamic type conversions are allowed (but all objects are typed).
5. Routine scheduling is explicit both with respect to calls to other routines and with respect to voluntary blocking, such as in buffer operations.
6. All Gypsy operations are expressed as CML routines, calls to which are expressed in the same way as calls to machine language operations.

5.1.2 The Capability Machine -

The basic property of the Capability Machine is that all references to objects are via a capability. This includes integers, arrays, routines (including primitive machine operations), and buffers attached to external devices (called special buffers in Gypsy). Thus a reference in this section to an "object" is equivalent to a reference to a "capability for an object", regardless of where the object itself is located. The descriptions of the operations are in CML.

5.1.3 Primitive Data Objects -

The single primitive data type is integer, and several operations such as copy and arithmetic on integers are provided via the CML interpreter. The precision of a given integer object can be specified, thus permitting single bits, eight bit bytes, etc., to be denoted. The Gypsy scalar (enumerated) type is implemented by the appropriate integer subset, and characters are implemented as eight bit integer values.

5.1.4 Predefined Structured Types -

Vectors of arbitrary and perhaps different typed objects are predefined and operations on them are provided.

5.1.5 The Unit Definition Table -

The unit definition table is used by the Capability Machine both to create objects according to a given definition, and to reference components of those objects. The unit definition table is a vector of definitions, one definition for each Gypsy unit known to the machine.

5.1.5.1 Type Definitions -

A type definition contains a code body for the allocation manager for that type. For software objects, this code creates a new instance (that is, a variable) of that type. A type definition for a software object specifies the number of components in an object of that type, the indices in the Unit Definition Table of their types, the length of the object in terms of multiples of the size of the primitive data object, and perhaps the access restrictions.

If the type is parameterized, the number and types of the parameters are also stated. The actual parameters are contained in a part of objects declared of that type that is accessible only by the referencing operation (in order to access components) and to the structure modification operation (used by the allocation routines). Any or all of the parts of a type object (i.e., number of components, type of components, length of object) may be

parameterized, and the parameters may be dynamically modified, permitting very flexible treatment of composed objects.

For Gypsy arrays, all components are of the same type, and only one type need be specified. Gypsy records are treated as vectors of objects of differing type, and field designators are reduced to vector indices. There are entries in the Unit Definition Table for the primitive and predefined types.

5.1.5.2 Routine Definition -

A routine definition contains the number of parameters, the type and required access restrictions (e.g., read-only) of each, and the code body of a Gypsy routine. Capabilities (called "unit capabilities") for all type and routine definitions which can be referenced by the code are contained in the routine definition. These capabilities may, for example, permit declarations of variables of a given type but not the right to reference components. A routine definition also includes a separate code body which is used to create and schedule a concurrent new activation record of the Gypsy routine.

5.1.6 Capabilities -

A capability contains the type, access rights, and contents location field of the object it names.

1. The type field is an index into the Unit Definition Table, and specifies the type of object named by the capability.
2. The access rights field is a bit vector which specifies the operations granted by this capability on the named object.
3. The contents location field specifies where the object is located. This field may be as simple as a single bit stating that the object immediately follows the capability, or as complicated as an address on some external device.

5.1.7 Object Allocation -

An object of a given type is brought into the referencing scope of a routine by a call to the allocation routine for that type. This is done by a call to the type definition, in which is included the allocation routine. The effect of such a call is to place the object on top of the data stack. If the object is a primitive resource, the manager returns the object on top of the stack. If it is a software object, the managing routine creates a new instance of the type by allocating a block of storage, on the top of the data stack (described in the next section) of the current

activation record, of sufficient size to contain the desired object and of the desired type. "Allocate" is a nonprimitive operation on the data stack provided by the Capability Machine. It may be written in CML, and is distinguished only by possessing the capabilities needed to modify the storage area from which software objects are allocated.

Notice that whether an object is primitive or software is transparent to the routine using the object. If the object may be shared among several concurrently executing routines, the allocated object can only be accessed after successfully executing the attach operation, which grants exclusive access to the object.

5.1.8 Activation Records And Code Bodies -

Activation records are objects created using a routine definition in the Unit Definition Table, and may be created like any other variable. In addition to access to unit capabilities from the routine definition, activation records contain a condition register and two stacks: the control stack and the data stack.

5.1.8.1 The Control Stack -

The control stack is a stack of control objects. A control object contains a vector of instructions, which are routine calls and control objects, and a current instruction pointer into that

vector. The control objects implement compound ("begin .. end"), selection ("if" and "case"), and loop statements. A routine call is the name of routine to be executed. Interpretation of the control stack is described later.

5.1.8.2 The Data Stack -

The data stack is a stack of data objects. The data objects may contain data objects nested to an arbitrary depth. Routines called via the control stack operate on the data in the data stack. Associated with the data stack is a stack bottom pointer, located in the activation record. All references to the data stack are relative to this base, which is set in the routine call mechanism.

5.1.9 The Routine Interpreter -

The routine interpreter (roughly corresponding to a CPU) contains two registers, a Unit Definition Table register, which contains a capability for the Unit Definition Table, and the Current Activation Record register, which contains a capability for the activation record currently being interpreted. When a software routine activation is called, the code body, which is a compound control object, is placed on the control stack, and the parameters are placed on the data stack, and execution of the compound control object begins. The interpretation algorithm is described below for

a control object. In general, the action of the interpreter is to execute the instruction pointed to by the pointer in the top control object on the control stack. If it is a routine call, the specified routine is executed (a detailed description of the call mechanism is given later); if it is a control object, that object is placed on the top of the control stack, its pointer initialized to its first instruction, and execution continues. When the pointer exceeds the number of instructions in the control object, the control object is popped from the stack and execution of the control object now at the top of the control stack continues at whichever instruction is denoted by the instruction pointer in that control object. The implementation of the three control objects is determined by the operations which sequence the pointer.

If the control stack becomes empty, the routine is exited. The function of the exit operation depends on the state of the control stack, and is described later.

5.1.9.1 Compound Control Object -

The compound control object consists of a vector of instructions which are to be sequentially executed. The instruction pointer is incremented by one as each instruction is executed.

5.1.9.2 Selection Control Object -

The selection control object consists of a sequence of instructions which evaluate the selection expression, and a vector of objects for each of the alternatives. The first element of these objects is a label, and the rest are instructions. The control object first executes the selection expression, and on the basis of its value (left in a special object called "casevar"), sets the instruction pointer to the appropriate value, so that on the next instruction cycle, the instruction which follows the matching label will be executed. The last such vector contains a distinguished label which matches any value. After execution of the instructions in that vector, the pointer is set off the end of the instruction vector of the control object, so that the control object will be terminated.

5.1.9.3 Condition Handling -

The condition register is set by the SIGNAL and return operations. When the condition register is set, the interpreter sets the instruction pointer to the last instruction in the current control object, rather than to the normal next value for the control object. The last object of any control object is a WHEN object. A WHEN object consists of a selection compound statement which selects on the value of the condition register. The condition register is

cleared by the operation that copies the value of the condition register into casevar. The ELSE alternative of the selection structure handles any condition not explicitly matched by any other alternative. It may perform the "signal casevar" operation, which simply propagates the condition into the containing control object.

If the control stack becomes empty and the condition register is still set, the routine exits with that condition. The details of exit are described later.

5.1.9.4 Loop Control Object -

A loop statement consists of a vector of instructions followed by a label. The last instruction is a "repeat" operation, which causes the instruction pointer to be reinitialized. The "leavestruct" operation can take a label as a parameter. Execution of the leavestruct operation causes the instruction pointer to be set to the end of the compound object. If the label matches, or there is no label in the leavestruct operation, the object is popped off the control stack and normal execution resumes. If there is not a match, the control object is popped off the control stack, the instruction pointer set to the end of that object, and the test repeated, until a match is found. The action taken if the control stack becomes empty is described in section 5.1.12.

5.1.10 Capability Operations -

The following operations on capabilities are provided:

1. Access. This operation pushes the designated object onto the data stack. Parameters to this operation are an index specifying an object in the stack, and an index into that object. The latter object is called the "accessed object". The accessed object is then logically located both at the top of the data stack and in the location specified in the stack. That is, a change to the stack top (for example, as a var parameter) causes the same change in the accessed object. In terms of capabilities, the capability at the top of the stack and the capability for the accessed object have object ids which name the same object. The capability for the object being indexed must access rights which permit the referencing operation.
2. Change type. This operation changes the type of the designated object. That is, the type component of the capability is changed. A capability for this operation is possessed only by routines which allocate instances of software objects (i.e., variables). They need to build blocks of storage into objects of a particular type.
3. Change access rights. This operation changes the access rights of an object to a new value. The new value must be a subset of the old value.

4. Change structure. This operation changes the parameters of an object; for example, the type and number of the components. If instances of a type are to have static structures (like Gypsy records and arrays), capabilities for instances of such a type do not permit this operation. On the other hand, dynamic objects, like sequences, may be implemented with a flexible number of elements.

A routine may make arbitrary requirements concerning the type and access rights required of its parameters. Notice that the ability to call a routine can be denied by either of two means: (1) Do not give the calling routine a capability for the called routine, or (2) do not permit the calling routine to obtain adequate capabilities to use as parameters.

5.1.11 Shared Objects -

Two primitive operations provide mutually exclusive access to a given object shared by two or more concurrently executing activation records. The single parameter to the attach and release operations is a shared object. If the object is available, attach changes the access rights to permit access to the object; otherwise, the access rights are not changed. An operation which tests the access rights of a capability can be used to determine if

the attach succeeded. "Release" surrenders access to the shared object. No scheduling is implied by the attach and release operations.

5.1.12 The Routine Invocation Operations In CML -

The routine invocation operations in CML are call, start, suspend, and return.

5.1.12.1 Call Operation -

The form of the call operation is

<routine name> <parameter 1> ... <parameter n>

where <routine name> is the name of the routine to be called. The parameters are the indices (perhaps symbolic, in the assembly language) in the current data stack of the objects to be passed to the called routine.

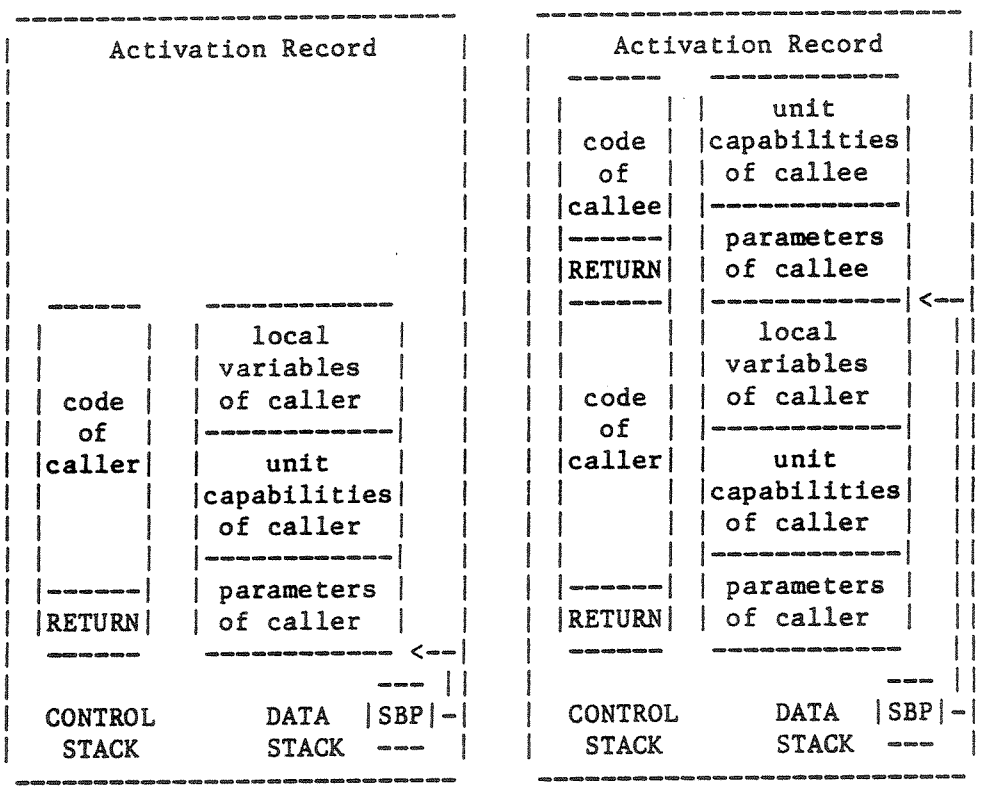
The call operation results in the execution of the named routine. For primitive routines, the interpreter passes the parameters to the routine and activates the routine. The caller is suspended by the interpreter until the primitive operation completes.

If <routine name> is a routine definition for a software routine (perhaps the image of a Gypsy routine), the interpreter pushes a return instruction onto the control stack, and on top of it pushes the code body of the called routine. See figure 5 for an example of a software routine call. The data stack bottom is set to the current value of the data stack top and the parameters are pushed onto it. The current stack bottom pointer is stored into the return instruction for later restoration. Thus, all references to the data stack are relative to the new base, and except for the parameters, the data of the caller is not accessible to the called routine. Onto the data stack the interpreter pushes the unit capabilities associated with that code body. Parameters to the routine are located beneath those capabilities. The interpreter then begins execution of the code body. The caller is thus implicitly suspended until the called routine exits.

Notice that the form of a call is identical regardless of whether the called routine is built into the machine, is a CML routine, or is the image of a Gypsy routine.

5.1.12.2 Start Operation -

The START operation states that execution of the routine which is its parameter is to begin, and the caller is to continue execution. It expects two parameters on the data stack: a routine



Before call
After return

After call

SBP = stack bottom pointer

Figure 5. Call/return of a software routine

name and a completion object. If <routine name> names a routine definition corresponding to a Gypsy routine unit, what is executed is a short CML routine which builds a new activation record on the data stack of the caller and schedules it for execution. This routine allocates a new activation record on the data stack, places the instruction and code body on the control stack of the new activation record, places the parameters and unit capabilities of that routine onto the new data stack, and calls a scheduling routine to place the activation record on the ready list. See figure 6 for the effect of a start operation on the caller's data stack.

5.1.12.3 Suspend Operation -

Shared between the caller and callee is a completion object. The suspend operation suspends its caller until something appears in this object. The completion object consists of two components: the name of the activation record which is suspended on it, and a count of how many routines have exited using this completion object. Part of the return operation is to reactivate the suspended routine.

After perhaps several routines have been started, the caller can suspend itself pending a condition object being placed in the completion object shared by the started routines; this is an implementation of the Gypsy cobegin statement. The suspend

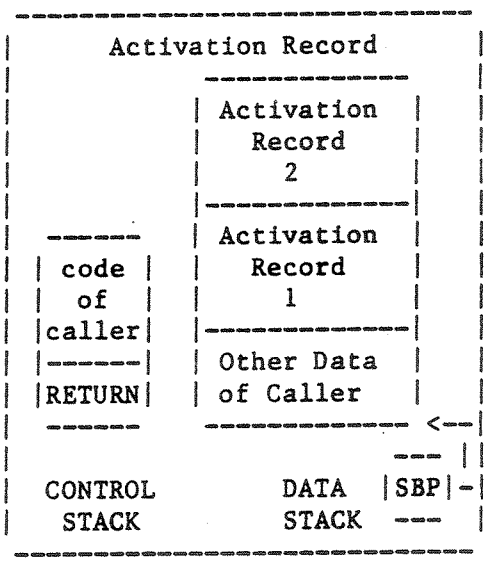


Figure 6. Effect of start operation

*

operation expects a completion object on the data stack as a parameter.

Another explicit use of the suspend operation is in blocking for Gypsy buffer operations. The routine desiring to be blocked arranges for an entry to be placed in a completion object when the needed buffer state is achieved (nonfull for send, nonempty for receive).

5.1.12.4 Return Operation -

When a primitive routine exits, the condition register is set by the interpreter to the appropriate value. When a called software routine completes, it executes the return operation (always the last instruction of a routine). If the control stack contains only the return instruction, the return operation places a condition object into the completion object. The control stack of the activation record is then empty and the activation record is deallocated. Setting the completion object includes resumption of the caller. When the caller is resumed, the condition object in the completion object is placed into the condition register of the calling routine, thus in effect signalling that condition in the caller. If the condition register of the exiting activation record contains a value corresponding to a condition parameter, or indicates no condition, that condition value is placed in the

condition object of the caller. The condition parameters contain condition objects known in the calling routine. If it is any other value, routinerror is placed into the caller's condition object.

If the return instruction is not the last instruction on the control stack, the return operation sets the condition register to a value as computed above, pops the data stack down to the stack bottom pointer, and then resets the bottom pointer to the value it had before the call to the returning routine. The return instruction is executed regardless of the value of the condition register. The calling routine thus cannot gain access to capabilities possessed by the called routine, even if the called routine returns in an error state. Notice that the parameters are popped off the data stack by the return operation. There is no explicit pop operation.

5.1.13 Routine Scheduling -

Process scheduling is centered around the CML interpreter and the ready queue. The CML interpreter is really just a device which interprets CML software routines. The ready queue is a priority ordered queue of activation records. There is a primitive CML operation called "next_routine" which causes the interpreter to load its current activation record register with the top entry of the ready queue, and thus begin interpretation of that routine.

This operation is executed as part of the suspend operation. On every instruction cycle, the CML interpreter polls a number of shared objects to see if something has been placed in them by an external device. The interpreter's reaction to such an event is to place the contents of the activation record register at the head of the ready queue and load the activation record register with the activation record of the routine which handles events on that shared object. One such shared object is set by the clock; the handler for this shared object may place the activation record at the head of the ready queue at the end of the ready queue and execute the next_routine operation. Notice that this polling is very similar to the test for interrupt algorithm performed in the microcode or logic of a conventional CPU.

The ready queue manager is responsible for ordering the ready queue. It takes as inputs requests from routines which start concurrent execution of activation records. These inputs include both the actual activation record to be executed, and various parameters concerning how the activation record is to be scheduled (e.g., a minimum service guarantee).

For external objects to communicate with the interpreter, and for routines to interact with the ready queue manager, they must both (1) have a capability for the appropriate shared object, and (2) possess the proper capabilities for objects to be placed into the shared object.

5.2 Design Issues And The Capability Machine Language

In this section, several major issues in the design of the architectural model are reviewed, along with how the Capability Machine reflects those considerations.

5.2.1 Storage Management -

1. Issue: All objects are referenced via capabilities, including simple objects such as integers.

   CM: All operations in CML, including "hardware" routines, use the same call mechanism. This mechanism places the parameters for the called routine onto the data stack; the parameters refer to objects accessible through the caller's data stack.

2. Issue: Objects may reside in various storage devices, which may be all or in part implemented in software.

   CM: Variables are allocated in "memory", special buffers are associated with external devices.

3. Issue: To permit capabilities to be very concise, type objects are defined, and capabilities may contain short references to the more lengthy descriptor fields contained in a type object.

CM: The type field of a capability is an index into the "unit definition table." The entries in that table contain type and routine definitions. For static objects, like Gypsy records, capabilities of components may be computed entirely from the type definition, and need not be located in the object.

4. Issue: The object named by a descriptor may either directly follow the descriptor (as with tags) or be located elsewhere (as with conventional capabilities); the operations which use the object cannot distinguish which is the case.

CM: The object id field of capabilities is thus defined.

5. Issue: An object may contain other objects; the contained objects are referenced by a naming convention associated with the particular storage object (i.e., an index). Implementation details, such as byte displacements of the components of an object, are not visible to the user of the object. Only the routines which possess the access rights to modify the structure of a given object are concerned with the actual layout of the components.

CM: All software objects, including the data stack, are treated like a vector of objects of perhaps differing

types. The referencing operation uses indices into the data stack and indices into objects on the stack; displacements are found from the type object or parameter part of the object (called the "header" in the model description).

5.2.2 Process Management -

1. **Issue: Software and hardware routines are treated uniformly.**

   CM: A routine calls another routine with the "call" operation. The form of the call operation is the same whether a hardware or software routine is called.

2. **Issue: The creation of activations of software routines is separated from calls on the activations.**

   CM: The sequential execution operation ("call") creates the execution environment for the called routine. The concurrent execution operation ("start") executes an activation record builder and scheduler, which is part of the routine definition.

3. Issue: The instructions in code objects are decoded by "dispatchers," which generate calls to the devices specified in the code. The dispatchers take the routines to be decoded from a ready queue.

   CM: The CM interpreter interprets code bodies by formulating calls to primitive and software routines.

5.2.3 Process Synchronization -

1. Issue: Device communication is via shared objects, on which synchronization primitives are provided. This technique is used as as the parameter passing mechanism.

   CM: The attach and detach operations are provided in CM. Sequential calls also pass information via shared objects (the parameters), but since the caller builds the execution environment of the called routine, the attach and detach operations are not needed to pass parameters for sequential calls.

2. Issue: The gaining of exclusive access to a shared object is separated from any scheduling which may be associated with failure to immediately gain the desired access.

CM: The attach and detach operations have this property. The suspend operation is provided to allow a routine to pause until an object appears in a given shared object.

5.3 Implementation Of The Capability Machine On Hardware

The Capability Machine is implemented on a PDP-11/03 (LSI-11) which is supplemented with a simple operating system called the "PROM Operating System." In this section, the PROM Operating System is briefly described, and then the major implementation issues. The term "runtime" in the following sections refers to execution on the PDP-11, rather than the "runtime" of the Cabability Machine.

5.3.1 The PROM Operating System -

The PROM Operating system is a very simple operating system implemented on a Programmable Read-Only Memory (thus its name). It provides three sets of virtual registers (and thus three processes, one for itself, two for the application), low level device drivers which handle all external interrupts, and an operator TTY interface. The operating system permits the application program to interact with it via the PDP-11 TRAP instruction.

5.3.2 Issues In The Capability Machine Implementation -

In this section several issues related to the Capability Machine implementation are discussed.

5.3.2.1 Capabilities -

The LSI-11 does not have memory management hardware, so runtime capability interpretation would be prohibitively expensive. The interaction of Gypsy routines with each other and with data objects is, by design, capable of mostly being done at parse time, by the semantic checker. It is the responsibility of the CML to PDP-11 object code generator to explicitly include code to check those things not checked by the parser (such as array bounds checks) and in general to interpret the capabilities (such as handling indirection).

5.3.2.2 Code Objects -

The code generator generates reasonable object code of operations directly supported by the target machine, such as arithmetic and boolean instructions. For example, operands are not pushed onto the PDP-11 stack to do an addition operation. Rather, a single instruction referencing the desired operands is generated. The Capability Machine primitives needed at runtime are written in PDP-11 assembly language.

5.3.2.3 The Control Stack -

The code generator maintains a stack which contains pointers to control structures which have been entered but not completed, due to a call in them to a nested control structure. Notice that the code generator image of the control stack is a very close analogue of the Capability Machine control stack. No runtime storage need be reserved for control objects in the runtime control stack; only routine linkage information is needed. The runtime control stack entries contain information to properly set the control and data stack for routine calls.

5.3.2.4 Data Stack -

The data stack in the implementation is much like the Capability Machine data stack. When a variable is allocated, space on the implementation data stack reserved for that variable. When the access operation is called, naming that variable, a pointer to the variable is pushed onto the data stack. Variables are always statically allocated, using the size restriction on dynamic objects to find the maximum amount of stack space needed. Formal parameters of software routines are pointers back to the actual parameters.

5.3.2.5 Conditions -

Conditions are forward jumps. A jump to a compile time known label which denotes the appropriate WHEN object is used to implement the signal and routine error returns. Code at that label resets the runtime data stack.

5.3.2.6 Concurrency -

Code objects in the implementation contain the code to generate and schedule a concurrent activation of that routine, just as in the Capability Machine. This code is bypassed on a sequential call to the routine. The support routines which manage the ready queue may, for the current implementation, schedule the ready routines in a round robin manner. The hardware clock will be used to activate the ready queue manager when a routine's quantum expires.

5.3.2.7 Efficiency -

The CML-to-PDP11 code generator generates PDP-11 instructions by processing the CML line by line. An effort has been made to generate as efficient code as possible, but in the current implementation, no attempt is made to optimize PDP-11 code which spans more than one CML instruction, other than some register management. The implementation methodology has been that the

generated PDP-11 code be functionally equivalent to the CML; most of the restrictions and descriptor interpretation operations required in the CM can be performed in the compilation steps and need not be visible as PDP-11 code.

5.3.2.8 Validation Of Model -

The Capability Machine turns out to be of great assistance in the implementation of Gypsy. It provides an intermediate step in the translation of Gypsy which is independent of the existing machine on which the code is to execute. The Capability Machine retains the desirable properties of typing, modularity, etc. The Gypsy compiler is very modular; the different parts of the compiler implement various parts of the functionality of the Capability Machine.

5.3.3 Implementation Stage -

As of May, 1978, about 80% of sequential Gypsy (that is, without cobegin statements and buffers) can be compiled and executed on an LSI-11. Work is continuing and more features of Gypsy are regulary made available. By July, 1978, essentially all of Gypsy, including concurrency, should be implemented. At that time, the compiler effort will turn primarily to optimization issues.

CHAPTER 6

CONCLUSION

Current hardware technology permits hardware to be designed which does a much better job of supporting modern software concepts than is common in contemporary architectures. This dissertation is intended to discuss how contemporary architectures interact with modern software techniques and to suggest ways of improving hardware support.

A primary feature of the proposed architectural model is the unification of the concepts of segment capabilities and word tags. All addressing, including simple objects like integers, is done via typed descriptors (capabilities). The type information is used to enable the descriptors of components of objects of that type to be expressed concisely.

The second primary feature of the model is the unification of the concepts of hardware operations, software routines, and hardware device calls. Mechanisms are provided which permit the use of a routine without having to take into account the implementation of the routine (for example, whether it is hardware or software).

The description of the model is essentially complete. Some refinement is needed on the basic operations on capabilities and headers. In particular, the new_object operation is rather vague with respect to what is to happen in a situation where two descriptors for the same object denote different types. The modify_header operation appears somewhat awkward, especially in that it implies changing the entire header in order to do something simple like creating a new component object.

The concept of controller needs to be clarified, especially with respect to the definition of the "function" part of a hardware device. How the controller-device concept works for realistic process scheduling problems needs to be further investigated.

The cost of implementing this model has not been computed. Clearly, one would not consider such an architecture if the primary concern was an absolute minimization of hardware cost. However, the advances in hardware technology and the demand for reliable but reasonably priced software favor implementing in hardware many features which support modern software concepts.

APPENDIX

THE CML IMAGE OF AN EXAMPLE GYPSY PROGRAM

The Gypsy routines Read_CML_Line, Get_Name, and Skip_Trailing are transliterations of the Pascal source of the lexical scanner in the CML-to-PDP11 code generator. Read_CML_Line reads lines of CML code into a structure of type CMLinstr. Generate writes these structures.

A CML line is of the form:

```
<CML_line>        ::= <label> <routine call>

<label>           ::= <label name> <separator>
                    | <space character>

<routine call>    ::= <operation name> <parameter list>
                    | <empty>

<parameter list>  ::= <parameter list> <parameter>
                    | <empty>

<parameter>       ::= <parameter name>
                    | <number>
                    | '<number>
```

<label name>, <parameter name> and <operation name> are strings not containing blanks, and <separator> is a non-blank character.

<number> is a numeric string, and is the CML representation of an integer literal. "'<number>" is the CML representation of the character literal whose ASCII value is <number>. Notice that labels must begin in the first column.

Some words are needed about some of the distinctive language features of Gypsy. The access lists (bracketed by "<>") on set_blank, Get_Name, and Skip_Trailing state that only Read_CML_Line can call them. The access list on CMLinstr states that only Read_CML_Line and generate know the definition (representation) of CMLinstr. This access list follows the unit name; all routines can declare variables and parameters of type CMLinstr.

The LOOP...END construct in Gypsy is an infinite loop. The LEAVE statement causes the most enclosing loop to be exited.

Gypsy has provisions for error handling. If Read_CML_Line detects a syntactic error in a line of CML, it signals the condition ill_formed_line. All control structures in Gypsy may end with a WHEN block, which looks like a CASE statement with condition names as case labels. Signalling an error condition causes a forward branch to the innermost enclosing WHEN part which contains a label which matches the signalled condition. If a routine boundary is found before a match is found, the condition (cond) parameters are examined. If the signalled condition is found in that list, the routine is exited and the condition name in the actual parameter list is signalled in the calling routine, and the search for a match

continues. If no match is found, the condition ROUTINE_ERROR is signalled at the call site. In the example, Read_CML_Line signals ill_formed_line if a syntactic error is found in a line of CML. The condition is not handled in Read_CML_Line (no condition is), so ill_formed is signalled in Main. The code in the WHEN part in procedure Main has the label ill_formed; it outputs an error message. The BEGIN-END structure in Main serves only to provide a place for a WHEN part; all Gypsy control structures take statement lists.

Several CML constructs need to be explained. The ACCESS operation places a descriptor for the accessed component onto the data stack. When a "*" appears as an operand, the accessed component at the top of the data stack is taken as the operand, and the descriptor is popped off the data stack.

All control structures are bracketed by BEGIN-END pairs. The LEAVESTRUCT operation says to exit the current control structure. LEAVESTRUCT LOOP says to exit the innermost control structure which has a "LOOP" identifier; that is, starts with "BEGIN LOOP" and ends with "END LOOP". The REPEAT operation causes a branch to the first statement of the current control structure. "CASE CASEVAR" causes a forward branch to a statement labelled with the value of CASEVAR, which is not nested in a deeper control structure. The ELSE label matches any value of the CASE variable.

```
type B1 = buffer (1) of character {input};
type B2 = buffer (1) of character {output};
type Gypsy_symbol = array[(1..20)] of character;

type CMLInstr <Read_CML_Line, generate> = record (
                    labelpart:Gypsy_symbol;
                    labelconn:character;
                    operation:Gypsy_symbol;
                    Noperands:int;
                    operandpart:array[(1..10)] of Gypsy_symbol
                  );

procedure get (var source:character; var ttyin:B1; var ttyout:B2) =
begin
  receive source from ttyin;
  send source to ttyout;
end;

procedure <read_CML_line> set_blank(var b: Gypsy_symbol) =
begin
  var i: int := 1;
  loop
    if i > 20 then leave end;
    b[i] := ' ' ; {' ' is the character literal for space}
    i := i + 1;
  end
end;

procedure <Read_CML_line> Get_Name(var name: Gypsy_symbol;
  var Source: character; var ttyin: B1; var ttyout: B2) =
begin
entry source ge ' ' ;
exit source ne ' ' ;
var i: int := 0;
  loop
    i:=i+1;
    name[i] := source;
    get (source, ttyin, ttyout);
    if source le ' ' then leave end;
  end;
  loop
    if source ne ' ' then leave end;
    get (source, ttyin, ttyout);
  end;
end;

procedure <Read_CML_Line> Skip_Trailing (var source: character;
  var ttyin: B1; var ttyout: B2) =
```

```
begin
  loop
    if source < ' '
    then get (source, ttyin, ttyout)
    else leave
    end
  end
end;

procedure Read_CML_Line(var Source: character; var ttyin: B1;
    var ttyout: B2; var Instr: CMLinstr; cond ill_formed_line) =
begin
entry  (source ge ' ') ;
exit (source ge ' ') ;
{ all characters less than ' '  (which includes CR and LF) are control
 characters }

  var blankname: Gypsy_symbol;
  set_blank(blankname);
  instr.LabelPart:=blankname;
  instr.LabelConn:= ' ';
  instr.Operation:=blankname;
  instr.Noperands:=0;
  loop
    if (instr.LabelPart ne blankname)
      or (instr.operation ne blankname) then leave end;
    Get_Name(instr.LabelPart, source, ttyin, ttyout);
    if instr.LabelPart ne blankname
    then
      if source < ' ' then signal ill_formed_line end;
      instr.LabelConn:=source;
      loop
        get (source, ttyin, ttyout);
        if source ne ' ' then leave end
      end;
    end;
    if source > ' '
    then
      Get_Name(instr.operation, source, ttyin, ttyout);
      loop
        if source le ' ' then leave end;
        instr.Noperands := instr.Noperands+1;
        instr.OperandPart[instr.Noperands]:=blankname;
        Get_Name(instr.OperandPart[instr.Noperands],
                 source, ttyin, ttyout);
      end;
    end;
    Skip_Trailing(source, ttyin, ttyout);
```

```
  end;
end;

procedure <generate> write_symbol(s: Gypsy_symbol; var ttyout: B2) =
begin
  var i: int:= 1;
  loop
    send s[i] to ttyout;
    i:= i+1;
    if i > 20 then leave end;
  end;
end;

procedure generate (instr: CMLinstr; var ttyout: B2) =
begin
  var i: int := 1;
  write_symbol (instr.LabelPart, ttyout);
  send instr.LabelConn to ttyout;
  write_symbol (instr.Operation, ttyout);
  loop
    if i > instr.Noperands then leave end;
    write_symbol(instr.OperandPart[i], ttyout);
    i := i + 1;
  end;
end;

procedure Main (var ttyin: B1; var ttyout: B2) =
begin
  var Instr: CMLinstr;
  var Source: character;
  cond ill_formed;

  get (source, ttyin, ttyout);
  loop
    begin
      Read_CML_Line (Source, ttyin, ttyout, Instr, ill_formed);
      generate (Instr, ttyout);
    when
      is ill_formed:
        send '# to ttyout; send 'i to ttyout; send 'l to ttyout;
        send 'l to ttyout; send '- to ttyout; send 'f to ttyout;
        send 'o to ttyout; send 'r to ttyout; send 'm to ttyout;
        send 'e to ttyout; send 'd to ttyout; send '  to ttyout;
        send 'l to ttyout; send 'i to ttyout; send 'n to ttyout;
        send 'e to ttyout; send '# to ttyout;
    end {begin}
  end {loop}
end; {main}
```

```
B1              = BEGINDESCRIPTOR
KIND            : BUFFER
ELEMENTS        : CHARACTER
SIZE            : 1
                  ENDDESCRIPTOR

B2              = BEGINDESCRIPTOR
KIND            : BUFFER
ELEMENTS        : CHARACTER
SIZE            : 1
                  ENDDESCRIPTOR

?T001X000       = BEGINDESCRIPTOR
KIND            : SUBRANGE
UPPERBND        : 20
LOWERBND        : 1
                  ENDDESCRIPTOR
GYPSY_SYMBOL    = BEGINDESCRIPTOR
KIND            : ARRAY
ELEMENTS        : CHARACTER
INDEX           : ?T001X000
                  ENDDESCRIPTOR

?T003X000       = BEGINDESCRIPTOR
KIND            : SUBRANGE
UPPERBND        : 10
LOWERBND        : 1
                  ENDDESCRIPTOR
?T002X000       = BEGINDESCRIPTOR
KIND            : ARRAY
ELEMENTS        : GYPSY_SYMBOL
INDEX           : ?T003X000
                  ENDDESCRIPTOR
CMLINSTR        = BEGINDESCRIPTOR
KIND            : RECORD
LABELPART       : GYPSY_SYMBOL
LABELCONN       : CHARACTER
OPERATION       : GYPSY_SYMBOL
NOPERANDS       : INT
OPERANDPART     : ?T002X000
                  ENDDESCRIPTOR


GET             = BEGINDESCRIPTOR
KIND            : PROCEDURE
BODY            : BEGIN
SOURCE          : RWPARAM          CHARACTER
TTYIN           : RWPARAM          B1
```

```
TTYOUT          : RWPARAM          B2
                  RECEIVE          SOURCE          TTYIN
                  SEND             SOURCE          TTYOUT
                  LEAVESTRUCT
                  END
                  ENDDESCRIPTOR

SET_BLANK       = BEGINDESCRIPTOR
KIND            : PROCEDURE
BODY            : BEGIN
B               : RWPARAM          GYPSY_SYMBOL
I               : ALLOCATE         INT
                  ASSIGN           I               1
                  BEGIN            LOOP
                  BEGIN
?T003X001       : ALLOCATE         BOOLEAN
                  GT               ?T003X001       I               20
                  ASSIGN           CASEVAR         ?T003X001
                  DEALLOCATE
                  CASE             CASEVAR
TRUE            :
                  LEAVESTRUCT      LOOP
                  LEAVESTRUCT
FALSE           :
                  LEAVESTRUCT
ELSE            : CMERROR
                  END

                  ACCESS           B               I
                  ASSIGN           *               '32
?T004X001       : ALLOCATE         INT
                  PLUS             ?T004X001       I               1
                  ASSIGN           I               ?T004X001
                  DEALLOCATE
                  REPEAT
                  END              LOOP
                  LEAVESTRUCT
                  END
                  ENDDESCRIPTOR

GET_NAME        = BEGINDESCRIPTOR
KIND            : PROCEDURE
BODY            : BEGIN
NAME            : RWPARAM          GYPSY_SYMBOL
SOURCE          : RWPARAM          CHARACTER
TTYIN           : RWPARAM          B1
TTYOUT          : RWPARAM          B2
I               : ALLOCATE         INT
```

```
                ASSIGN          I               0
                BEGIN           LOOP
?T006X001       : ALLOCATE      INT
                PLUS            ?T006X001       I               1
                ASSIGN          I               ?T006X001
                DEALLOCATE

                ACCESS          NAME            I
                ASSIGN          *               SOURCE
                GET             SOURCE          TTYIN           TTYOUT
                BEGIN
?T007X001       : ALLOCATE      BOOLEAN
                LE              ?T007X001       SOURCE          '32
                ASSIGN          CASEVAR         ?T007X001
                DEALLOCATE
                CASE            CASEVAR
TRUE            :
                LEAVESTRUCT     LOOP
                LEAVESTRUCT
FALSE           :
                LEAVESTRUCT
ELSE            : CMERROR
                END
                REPEAT
                END             LOOP
                BEGIN           LOOP
                BEGIN
?T008X001       : ALLOCATE      BOOLEAN
                NENUM           ?T008X001       SOURCE          '32
                ASSIGN          CASEVAR         ?T008X001
                DEALLOCATE
                CASE            CASEVAR
TRUE            :
                LEAVESTRUCT     LOOP
                LEAVESTRUCT
FALSE           :
                LEAVESTRUCT
ELSE            : CMERROR
                END
                GET             SOURCE          TTYIN           TTYOUT
                REPEAT
                END             LOOP
                LEAVESTRUCT
                END
                ENDDESCRIPTOR

SKIP_TRAILING   = BEGINDESCRIPTOR
KIND            : PROCEDURE
```

```
BODY            : BEGIN
SOURCE          : RWPARAM       CHARACTER
TTYIN           : RWPARAM       B1
TTYOUT          : RWPARAM       B2
                  BEGIN         LOOP
                  BEGIN
?T004X001       : ALLOCATE      BOOLEAN
                  LT            ?T004X001       SOURCE          '32
                  ASSIGN        CASEVAR         ?T004X001
                  DEALLOCATE
                  CASE          CASEVAR
TRUE            :
                  GET           SOURCE          TTYIN           TTYOUT
                  LEAVESTRUCT
FALSE           :
                  LEAVESTRUCT   LOOP
                  LEAVESTRUCT
ELSE            : CMERROR
                  END
                  REPEAT
                  END           LOOP
                  LEAVESTRUCT
                  END
                  ENDDESCRIPTOR

READ_CML_LINE   = BEGINDESCRIPTOR
KIND            : PROCEDURE
BODY            : BEGIN
SOURCE          : RWPARAM       CHARACTER
TTYIN           : RWPARAM       B1
TTYOUT          : RWPARAM       B2
INSTR           : RWPARAM       CMLINSTR
ILL_FORMED_LINE : READPARAM      CONDITION
BLANKNAME       : ALLOCATE      GYPSY_SYMBOL
                  SET_BLANK     BLANKNAME

                  ACCESS        INSTR           LABELPART
                  ASSIGN        *               BLANKNAME

                  ACCESS        INSTR           LABELCONN
                  ASSIGN        *               '32

                  ACCESS        INSTR           OPERATION
                  ASSIGN        *               BLANKNAME

                  ACCESS        INSTR           NOPERANDS
                  ASSIGN        *               0
                  BEGIN         LOOP
```

```
                BEGIN
?T007X001       : ALLOCATE      BOOLEAN
?T008X002       : ALLOCATE      BOOLEAN

                  ACCESS        INSTR           LABELPART
                  NE            ?T008X002   *               BLANKNAME
?T009X003       : ALLOCATE      BOOLEAN

                  ACCESS        INSTR           OPERATION
                  NE            ?T009X003   *               BLANKNAME
                  OR            ?T007X001   ?T008X002       ?T009X003
                  DEALLOCATE
                  DEALLOCATE
                  ASSIGN        CASEVAR         ?T007X001
                  DEALLOCATE
                  CASE          CASEVAR
TRUE            :
                  LEAVESTRUCT   LOOP
                  LEAVESTRUCT
FALSE           :
                  LEAVESTRUCT
ELSE            : CMERROR
                  END

                  ACCESS        INSTR           LABELPART
                  GET_NAME  *         SOURCE      TTYIN       TTYOUT
                  BEGIN
?T010X001       : ALLOCATE      BOOLEAN

                  ACCESS        INSTR           LABELPART
                  NE            ?T010X001   *               BLANKNAME
                  ASSIGN        CASEVAR         ?T010X001
                  DEALLOCATE
                  CASE          CASEVAR
TRUE            :
                  BEGIN
?T011X001       : ALLOCATE      BOOLEAN
                  LT            ?T011X001       SOURCE          '32
                  ASSIGN        CASEVAR         ?T011X001
                  DEALLOCATE
                  CASE          CASEVAR
TRUE            :
                  SIGNAL        ILL_FORMED_LINE
                  LEAVESTRUCT
FALSE           :
                  LEAVESTRUCT
ELSE            : CMERROR
                  END
```

```
                ACCESS          INSTR           LABELCONN
                ASSIGN          *               SOURCE
                BEGIN           LOOP
                GET             SOURCE          TTYIN           TTYOUT
                BEGIN
?T012X001       : ALLOCATE      BOOLEAN
                NENUM           ?T012X001       SOURCE          '32
                ASSIGN          CASEVAR         ?T012X001
                DEALLOCATE
                CASE            CASEVAR
TRUE            :
                LEAVESTRUCT     LOOP
                LEAVESTRUCT
FALSE           :
                LEAVESTRUCT
ELSE            : CMERROR
                END
                REPEAT
                END             LOOP
                LEAVESTRUCT
FALSE           :
                LEAVESTRUCT
ELSE            : CMERROR
                END
                BEGIN
?T013X001       : ALLOCATE      BOOLEAN
                GT              ?T013X001       SOURCE          '32
                ASSIGN          CASEVAR         ?T013X001
                DEALLOCATE
                CASE            CASEVAR
TRUE            :

                ACCESS          INSTR           OPERATION
                GET_NAME  *           SOURCE     TTYIN          TTYOUT
                BEGIN           LOOP
                BEGIN
?T014X001       : ALLOCATE      BOOLEAN
                LE              ?T014X001       SOURCE          '32
                ASSIGN          CASEVAR         ?T014X001
                DEALLOCATE
                CASE            CASEVAR
TRUE            :
                LEAVESTRUCT     LOOP
                LEAVESTRUCT
FALSE           :
                LEAVESTRUCT
ELSE            : CMERROR
```

```
                  END
?T015X001      : ALLOCATE        INT
                  ACCESS         INSTR          NOPERANDS
                  PLUS           ?T015X001      *              1

                  ACCESS         INSTR          NOPERANDS
                  ASSIGN         *              ?T015X001
                  DEALLOCATE

                  ACCESS         INSTR          OPERANDPART

                  ACCESS         INSTR          NOPERANDS
                  ACCESS         *              *
                  ASSIGN         *              BLANKNAME

                  ACCESS         INSTR          OPERANDPART

                  ACCESS         INSTR          NOPERANDS
                  ACCESS         *              *
                  GET_NAME   *         SOURCE     TTYIN      TTYOUT
                  REPEAT
                  END            LOOP
                  LEAVESTRUCT
FALSE          :
                  LEAVESTRUCT
ELSE           : CMERROR
                  END
                  SKIP_TRAILING  SOURCE         TTYIN          TTYOUT
                  REPEAT
                  END            LOOP
                  LEAVESTRUCT
                  END
                  ENDDESCRIPTOR

WRITE_SYMBOL   = BEGINDESCRIPTOR
KIND           : PROCEDURE
BODY           : BEGIN
S              : READPARAM       GYPSY_SYMBOL
TTYOUT         : RWPARAM         B2
I              : ALLOCATE        INT
                  ASSIGN         I              1
                  BEGIN          LOOP

                  ACCESS         S              I
                  SEND           *              TTYOUT
?T004X001      : ALLOCATE        INT
                  PLUS           ?T004X001      I              1
```

```
                    ASSIGN          I               ?T004X001
                    DEALLOCATE
                    BEGIN
?T005X001       :   ALLOCATE        BOOLEAN
                    GT              ?T005X001       I               20
                    ASSIGN          CASEVAR         ?T005X001
                    DEALLOCATE
                    CASE            CASEVAR
TRUE            :
                    LEAVESTRUCT     LOOP
                    LEAVESTRUCT
FALSE           :
                    LEAVESTRUCT
ELSE            :   CMERROR
                    END
                    REPEAT
                    END             LOOP
                    LEAVESTRUCT
                    END
                    ENDDESCRIPTOR

GENERATE        =   BEGINDESCRIPTOR
KIND            :   PROCEDURE
BODY            :   BEGIN
INSTR           :   READPARAM       CMLINSTR
TTYOUT          :   RWPARAM         B2
I               :   ALLOCATE        INT
                    ASSIGN          I               1

                    ACCESS          INSTR           LABELPART
                    WRITE_SYMBOL    *               TTYOUT

                    ACCESS          INSTR           LABELCONN
                    SEND            *               TTYOUT

                    ACCESS          INSTR           OPERATION
                    WRITE_SYMBOL    *               TTYOUT
                    BEGIN           LOOP
                    BEGIN
?T004X001       :   ALLOCATE        BOOLEAN

                    ACCESS          INSTR           NOPERANDS
                    GT              ?T004X001       I               *
                    ASSIGN          CASEVAR         ?T004X001
                    DEALLOCATE
                    CASE            CASEVAR
TRUE            :
                    LEAVESTRUCT     LOOP
```

```
                LEAVESTRUCT
FALSE         :
                LEAVESTRUCT
ELSE          : CMERROR
                END

                ACCESS          INSTR           OPERANDPART
                ACCESS          *               I
                WRITE_SYMBOL    *               TTYOUT
?T005X001     : ALLOCATE        INT
                PLUS            ?T005X001       I               1
                ASSIGN          I               ?T005X001
                DEALLOCATE
                REPEAT
                END             LOOP
                LEAVESTRUCT
                END
                ENDDESCRIPTOR

MAIN          = BEGINDESCRIPTOR
KIND          : PROCEDURE
BODY          : BEGIN
TTYIN         : RWPARAM         B1
TTYOUT        : RWPARAM         B2
INSTR         : ALLOCATE        CMLINSTR
SOURCE        : ALLOCATE        CHARACTER
ILL_FORMED    : ALLOCATE        CONDITION
                GET             SOURCE          TTYIN           TTYOUT
                BEGIN           LOOP
                BEGIN
                READ_CML_LINE SOURCE TTYIN TTYOUT INSTR  ILL_FORMED
                GENERATE        INSTR           TTYOUT
                LEAVESTRUCT
                WHEN
                CONDITION_VALUE CASEVAR         CAR
                CASE            CASEVAR
ILL_FORMED    :
                SEND            '35             TTYOUT
                SEND            '73             TTYOUT
                SEND            '76             TTYOUT
                SEND            '76             TTYOUT
                SEND            '45             TTYOUT
                SEND            '70             TTYOUT
                SEND            '79             TTYOUT
                SEND            '82             TTYOUT
                SEND            '77             TTYOUT
                SEND            '69             TTYOUT
                SEND            '68             TTYOUT
```

```
                SEND            '32             TTYOUT
                SEND            '76             TTYOUT
                SEND            '73             TTYOUT
                SEND            '78             TTYOUT
                SEND            '69             TTYOUT
                SEND            '35             TTYOUT
                LEAVESTRUCT
ELSE            :
                SIGNAL          CASEVAR
                END
                REPEAT
                END             LOOP
                LEAVESTRUCT
                END
                ENDDESCRIPTOR
```

REFERENCES

1. [Ambler 76] Ambler, A. L., D. I. Good, W. F. Burger. "Report on the Language Gypsy", ICSCA-CMP-1, The University of Texas at Austin, 1976.

2. [Amdahl 64] Amdahl, G. M. "The Structure of System/360, Part III - Processing Unit Design Considerations", IBM Sys. Journal 3, 2, 1964, pp. 144-164.

3. [Bell 73] Bell, James R. "Threaded Code", CACM 16, 6 (June, 1973) pp. 370-372.

4. [Brinch Hansen 70] Brinch Hansen, P. "The Nucleus of a Multiprogramming System", CACM 13, 4 (April 1970), pp. 238-250.

5. [Brinch Hansen 73] Brinch Hansen, P. Operating Systems Principles, Prentice-Hall, 1973.

6. [Cohen 75] Cohen, E., D. Jefferson. "Protection in the Hydra Operating System", Operating System Review 9, 5, 1975, pp. 141-161.

7. [Dahl 66] Dahl, O. J., K. Nygaard. "SIMULA - An ALGOL-Based Simulation Language", CACM 9, 9 (Sept. 1966), pp. 671-678.

8. [DeMartinis 76] DeMartinis, M. "A Self Managing Secondary Memory System", Proc. 3rd Annual Symp. on Computer Architecture, January, 1976.

9. [Dennis 65] Dennis, J. B. "Segmentation and Design of Multiprogrammed Computer Systems", J. ACM 12, 4 (Oct. 1965) pp. 589-602.

10. [Dennis 76] Dennis, J. "Computer Architecture and the Cost of Software" Computer Architecture News 5, 1 (April 1976), pp. 17-21.

11. [Digital 73] Digital Equipment Corporation. "PDP-11/45 Processor Handbook", Digital Equipment Corporation, Maynard, Massachusetts, 1973.

12. [Dijkstra 68] Dijkstra, E. "Cooperating Sequential Processes", Programming Languages, Academic Press, 1968.

13. [England 72] England, D. M. "Architectural Features of System 250", Plessey Telecommunications Ltd., 1972.

14. [Fabry 74] Fabry, R. S. "Capability-based Addressing", CACM 17, 7, July, 1974.

15. [Feustal 72] Feustal, E. A. "The Rice Research Computer - A Tagged Architecture", AFIPS Conf. Proc. vol. 40, 1972 SJCC, pp. 369-378.

16. [Feustal 73] Feustal, E. A. "On the Advantages of Tagged Architecture", IEEE Transactions on Computers, C-22 7 (July 1973), pp. 644-556.

17. [Ford 76] Ford, W. S., V. C. Hamacher. "Hardware Support for Inter-process Communication and Processor Sharing", Third Symposium on Computer Architecture, Jan. 1976, Comp. J. 20, 1977, pp. 155-162.

18. [Good 77] Good, D. I., ed. "Constructing Verifiably Reliable and Secure Communications Processing Systems", ICSCA-CMP-6, University of Texas at Austin, January, 1977.

19. [Hamer-Hodges 72] Hamer-Hodges, K. J. "Fault Resistance and Recovery within System 250", Intern. Conf. on Computer Communication, Oct. 1972.

20. [Hauck 68] Hauck, E. A., B. H. Dent. "The Burroughs B6500/7500 Stack Mechanism", AFIPS Conf. Proc., 1968 SJCC.

21. [Hoare 72] Hoare, C. A. R. "Notes on Data Structuring", Structured Programming, Academic Press, 1972.

22. [Hoare 74] Hoare, C. A. R. "Monitors - an Operating System Structuring Concept", CACM 17, 10 (Oct. 1974), pp. 549-557.

23. [IBM 68] IBM System/360 PL/I Reference Manual. SRL Form C28-8201-1, IBM Corp., 1968.

24. [Iliffe 69] Iliffe, J. K. "Elements of BLM", Computer J. 12 (Aug. 1969) pp. 251-258.

25. [Jones 75] Jones, A., R. Lipton. "The Enforcement of Security Policies for Computation", Operating System Review 9, 5 (1975), pp. 197-206.

26. [Lampson 73] Lampson, B. W. "A Note on the Confinement Problem", CACM 16, 10 (Oct. 1973).

27. [Levin 75] Levin, R., E. Cohen, W. Corwin, F. Pollack, W. Wulf. "Policy/Mechanism Separation in Hydra", Operating System Review 9, 5, (1975) pp. 132-140.

28. [Liskov 74] Liskov, B.,S. Zilles. "Programming With Abstract Data Types", Proceedings of ACM SIGPLAN Conference on Very High Level Languages, SIGPLAN Notices 9, 4 (April 1974), pp. 50-59.

29. [Lonergan 61] Lonergan, W., P. King. "Design of the B5000 System", Datamation 7, 5 (May 1961), pp. 28-32.

30. [Neumann 75] Neumann, P. G., L. Robinson, K. N. Levitt, R. S Boyer, A. R. Saxena. A provably secure operating system, Stanford Research Inst. Final Report, Menlo Park, Calif., June, 1975.

31. [Parnas 72] Parnas, D. L. "A Technique for Software Module Specification with Examples", CACM 15, 5 (May 1972), pp. 330-336.

32. [Randell 75] Randell, B. "System Structure for Software Fault Tolerance", Proc. International Conf. on Reliable Software, SIGPLAN Notices 10, 6, 1975, pp. 437-449.

33. [Redell 74] Redell, D., R. Fabry. "Selective Revocation of Capabilities", International Workshop on Protection in Operating Systems, IRIA, 1974.

34. [Repton 72] Repton, C. S. "Reliability Assurance for System 250. A Reliable, Real-Time Control System", Intern. Conf. on Computer Communication, Washington, D. C., Oct. 1972.

35. [Rice 68] "Rice University Computer - Basic Machine Operation", Rice University, Houston, Texas, Revised 1968.

36. [Saltzer 74] Saltzer, J. H. "Protection and the control of information sharing in Multics", CACM 17, 7 (July 1974), pp. 388-402.

37. [Schroeder 72] Schroeder, M. Cooperation of Mutually Suspicious Subsystems in a Computer Utility, Ph.D. thesis, Massachusetts Institute of Technology, MAC TR-104, Sept. 1972.

38. [Texas 76] Texas Instruments, Inc. "990 Computer Family Systems Handbook", Manual no. 945250-9701, Austin, Texas, 1976.

39. [TR440 70] "TR440 Eigenshaften des RD441", AEG TELEFUNKEN, Konstanz, Germany, Rep DBS 180 0470, March 1970.

40. [Watson 72] Watson, W. J. "The TI ASC - a highly modular and flexible super computer architecture", AFIPS Conf. Proc., 1972 FJCC 41 (part I), pp. 221-228.

41. [Xerox 70] Xerox Data Systems. "XDS Sigma 9 Computer", El Segundo, California, Publ. 90 17 33A, Oct. 1970.

VITA

Charles George Hoch was born in San Antonio, Texas on June 9, 1950, the son of Ida George Hoch and Charles Floyd Hoch. He was graduated from Robert E. Lee High School, San Antonio, Texas, in 1968, and entered Texas Lutheran College at Seguin, Texas in that year. He received the degree of Bachelor of Science with a major in chemistry from Texas Lutheran College in May 1972. In September 1972, he entered the Graduate School of The University of Texas at Austin, in the Department of Chemistry. In January 1973, he moved to the Department of Computer Science at the same university. In December 1973, he married Debra Kay Nance of Austin. Since 1975 he has been working as a research assistant for the Certifiable Minicomputer Project at The University of Texas at Austin.

Permanent address: 2217 S. Lakeshore Blvd., Apt. 210

Austin, Texas 78741