26742

National Library   Bibliothèque nationale   CANADIAN THESES   THÈSES CANADIENNES
of Canada   du Canada   ON MICROFICHE   SUR MICROFICHE

NAME OF AUTHOR NOM DE L'AUTEUR _____

TITLE OF THESIS TITRE DE LA THÈSE _____

_____

_____

UNIVERSITY UNIVERSITÉ _____

DEGREE FOR WHICH THESIS WAS PRESENTED
GRADE POUR LEQUEL CETTE THÈSE FUT PRÉSENTÉE _____

YEAR THIS DEGREE CONFERRED ANNÉE D'OBTENTION DE CE GRADE _____

NAME OF SUPERVISOR NOM DU DIRECTEUR DE THÈSE _____

DATED DATE _____ SIGNED SIGNÉ _____

PERMANENT ADDRESS RÉSIDENCE FIXE _____

_____

_____

NL-91 3-74

THE UNIVERSITY OF ALBERTA

PRINCIPLES OF MULTIPLE CONCURRENT COMPUTER EMULATION

by

Ⓒ John Charles Demco

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

FALL, 1975

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and
recommend to the Faculty of Graduate Studies and Research, for
acceptance, a thesis entitled "Principles of Multiple
Concurrent Computer Emulation", submitted by John Charles
Demco in partial fulfilment of the requirements for the degree
of Master of Science.

....
Supervisor

....

....

Date ...August 11, 1975....

ABSTRACT

Emulation of a computer or process using a microprogrammable machine has widespread application, as has the use of a computer to handle a number of tasks concurrently. This thesis seeks to combine these two concepts. An environment is proposed whereby a control program (not unlike a multiprogramming supervisor, but entirely microprogrammed) manages a number of processes, typically computer emulators. Principles incorporated in the design and implementation of such a control program are discussed, and a description of the major components of an actual implementation is given.

iv

ACKNOWLEDGMENTS

TABLE OF CONTENTS

vi

LIST OF FIGURES

Introduction

With the advent of writable control stores and user-microprogrammable computers, microprogramming is becoming more and more a research tool rather than simply a building tool. One of the fields of research in microprogramming which is enjoying prominence is the emulation of computers and of high level language processors. This thesis seeks to extend the concepts involved in providing support for emulators by considering multiple concurrent computer emulation. Specifically, a compact set of principles governing multi-emulator system design will be presented, and components of an actual implementation will be described.

A number of parallels to the development of multi-emulation may be drawn in terms of user support, supervisor complexity, and architectural complexity. From the point of view of user support, consider the transition from single-user stand-alone programming to single user operating systems to multi-user operating systems. In terms of supervisor complexity, consider the extension from support of single users to the support of many users to the support of many operating systems concurrently (e.g. see IBM Corporation (1972) for an introduction to their multi-operating system

facility VM/370). Finally, from the architectural point of view, increasing complexity is evident in the development from single processor machines to multi-processor machines to computer networks. Each of these sequences bears similarity to the following transition sequence: stand-alone computer emulators to "integrated" emulators (Allred, 1971) to multiple emulations. The increasing degree of complexity and potential is apparent in each.

Although Tucker (1965) introduced the term, Rosin's (1969) definition of emulator will be used: "a complete set of microprograms which, when embedded in a control store, define a machine." (This "machine" is usually a computer or a high level language processor.) Tucker's definition of emulation refers primarily to software routines and hardware modifications, and is inappropriate in a microprogramming context. A virtual machine is the machine realized by an emulator, and a host machine is one which supports these microprograms. This work will not deal with microprogramming itself; the reader may refer, for example, to Husson (1970). Multiple emulation or multi-emulation is the concurrent execution of a (usually hetereogeneous) set of emulators on a single host machine under the control of a microprogrammed supervisor; each emulator in the set is referred to as a member emulator. A reasonable analogy is a multiprogramming environment with a varied job mix.

In fact, many principles applied in multiprogramming may

also be applied in multi-emulation. The basic concepts of
memory, processor, device, and information management may be
found in a text such as Madnick and Donovan (1974).

It is important to note that this thesis will be
concerned for the most part with an environment in which the
member emulators are computers rather than language
translators. The resultant differences in the multi-emulator
supervisor are profound: this is due primarily to the great
difference in the level of the corresponding input/output
interfaces. The specification that the system support exact
emulations of computers cannot be overstressed; it is easily
the most important single factor affecting multi-emulator
supervisor design. A contemporary computer carries out I/O
operations at a low level: it deals in status bits, timing
considerations, interrupts, and the like. Communication is
highly device dependent. The simplest and perhaps only way for
the supervisor to provide this sort of communication is to let
the emulators interface with I/O devices directly. On the
other hand, most high level language processors are not
concerned with these matters at all: I/O is accomplished at a
high level and is usually quite device independent. Reliance
is placed on the supervisor to handle the actual low level
communication.

There are several reasons for the examination of a
multi-emulator environment consisting of emulators of
traditional computers rather than of language processors:

- There is still much interest in supporting computer emulators; a low level interface is therefore desirable.

- Provision of a high level device-independent interface nevertheless requires actual communication with devices at some point: it is reasonable therefore to start by providing this low level support initially, working up to the design and implementation of a high level interface.

The following chapter is an overview of multi-emulation, presenting a number of possible applications of multi-emulator systems along with a look at related work in the field. Chapter III is a consideration of the design of a multi-emulator supervisor, including required and desirable supervisor functions, a small set of underlying design principles, and a list of guidelines for the designer and implementor. Chapter IV applies these guidelines in the description of a particular multi-emulator supervisor for a C*-* computer: Finally, results and several suggestions for further research are stated in Chapter V.

An Overview of Multi-Emulation

2.1 Introduction

The purpose of this chapter is provide justification for the study of multi-emulation in its own right. A number of possible applications of a multi-emulator environment are given in the following section. These are in addition to the obvious argument that the extension from a multiprogramming environment to a multiple operating system environment to a multiple computer environment is a worthwhile effort because it is the extension of a concept. Section 2.2 presents descriptions of related work in this field. This review is necessarily short, as very little work has been done on multi-emulation.

2.2 Applications

Given a multi-emulator environment, a wide range of applications is possible. The applications suggested below are primarily directed toward emulator designers and implementors; however, some other novel uses are mentioned. Historically, there have been a number of reasons for writing emulators: emulation of an outdated machine to retain the use of software on which a great deal of money has been spent; emulation of an

experimental machine when sufficient building funds are not available, final design specifications have not been made, or architectural flexibility is desired; and emulation of processes, usually high level machine-independent language processors. In addition, research has been carried out on high level microprogramming languages for both vertical and horizontal control structures. For example, see Husson (1970), Ramamoorthy and Tsuchiya (1974), Eckhouse (1973), and Dasgupta (1974). Lloyd and Van Dam (1974) consider the problem as well and provide a good list of references. There is even interest in microprogrammed operating systems. See Huberman (1970) and Liskov (1972) for a discussion of the Venus operating system, and Werkheiser (1970) for a more general discussion. All of these capabilities are of course included in a multi-emulator environment. Here are some others:

• Providing that an inter-emulator communication mechanism were set up, a restricted form of network simulation could be produced. Probably the most severe problem would be that of synchronization, as the use of a single processor does not usually permit execution of parallel events, at least not on a macroscopic scale. This idea generates two related ideas which follow.

• Research could be performed on computer interfacing problems which are not specifically network problems. For example, in a simulated computer/front end environment, protocol could

be established and software written and tested at both ends. Again, synchronization problems would exist. As another example, a special I/O or processing device could be designed and debugged without actually existing. For instance, Dalrymple (1972) describes a microprogrammed, virtual associative memory which was added to an IBM 1130 emulation.

- Another related possibility is actual interface of the host machine to other real computers by having the real computer communicate with an emulator of the same type of computer. This would be done on the premise that it is easier for a computer to communicate with one of its own kind rather than another. Design of the actual interface hardware might not be any simpler, but the writing and testing of software to support the link might.

- A very interesting prospect is the implementation of an emulator the purpose of which is to function as a debugging and developmental aid to the emulator implementor. (Of course, the multi-emulator supervisor should itself provide rudimentary debugging and tracing facilities. See section 3.2.) The debugging emulator would have a mechanism to pass control to the unfinished emulator; the unfinished emulator would be temporarily fitted with a mechanism to return control to the debugger, along with pertinent data in a communications area. (Something along these lines has been

done by Dalrymple and Durakovich (1974). See section 2.3.)
The debugger would also have the ability to examine and
modify the unfinished emulator's storage.

* The above application is an important specific instance of a
more general application, that of machine hierarchies. This
concept involves two emulators operating in a master-slave
relationship so that the master can control the slave in
order to debug it, monitor it, gather run-time statistics
about it, or whatever. Of course, a significant time
degradation would occur. If one chooses, one may regard this
as a special type of network. Implementation of such a
scheme involves fairly sophisticated communication between
the two emulators, and if the slave is to be ignorant of the
master's existence (so as not to bias results) then the
supervisor should handle all of the communication. The idea
of supervisor interposition is important for other reasons,
as we shall see later.

* The existence of a multi-emulator system would make easier
the writing and implementing of high level language
machines. Many designs for such machines call for a two-
phase process, the first phase translating input text into
some more compressed intermediate form and the second
interpreting this intermediate text. For example, Melbourne
and Pugmire (1965) describe a FORTRAN machine, Weber (1967)
presents an implementation of an EULER machine, and Hassitt,

Lageschulte, and Lyon (1973) give details of an APL machine implementation on an IBM 360/25. Such a setup could fairly easily be implemented, given a multi-emulator environment. As well, the multi-emulator supervisor could include a set of (firmware) routines to carry out high level I/O processing for member emulators. The idea of high level I/O at the microprogram level is very interesting and is discussed in more detail in section 3.5.

2.3 Related Work

Little research has been done on multi-emulation. Rosin (1969) speculates briefly on it, first in relation to a high level language environment consisting of pairs of emulators, one for compilation and one for interpretation of a given language:

> The result could indeed be, for example, a FORTRAN machine having the potential advantages of both compilation and interpretation. The chief drawback is that a new emulator would have to be built for each language environment to be emulated. In a multiprogramming environment this could lead to requirements for a very large control store or a most efficient paging scheme for swapping emulators. A further disadvantage is that such emulators would have to be rewritten to take full advantage of the specific internal organization of all machines involved.

Another approach to high level language support he suggests is the building of general purpose emulators, implying the existence of one intermediate target language for all translators. A similar approach has been taken in the Burroughs B1700 system, to be discussed shortly.

Posin .also raises some questions on system protection and integrity:

If a system is to support more than one emulator, either simultaneously or using overlay techniques, questions of the following sort arise:

-Using an example, when returning from SNOBOL4 emulation to (say) 360 emulation, how does one assure that he will get back?

-In the same situation, how does one know there is something to get back to (the 360 emulator roll-out area may have been accidentally destroyed)?

-Does the system have to wait until all I/O operations initiated under one emulator reach completion before calling in another?

-How can information be passed between emulators?

-How are files to be formulated for emulator independent processing?

-How are interrupts handled (or emulated) and by which emulator?

The questions are not new, only the context has changed. They are quite analogous to the problems which led to and arose during the development of contemporary operating systems.

Rosin then goes on to mention the interesting tradeoff of allowing an emulator to use the hardware directly (and relying on the emulator for system integrity) versus degrading system performance by applying protection control (see section 2.3.1).

The remainder of this section gives short summaries of existing and proposed multi-emulator systems, and of systems which bear similarity to multi-emulation. These summaries provide a good overview of the present state of multi-emulation.

• Hopkins - batched multi-emulation

    Hopkins (197?) speculates about the design of a multi-emulator operating system on a microprogrammable computer with a writable control store, envisioning a single-thread sequential batch processing environment with only one emulator resident in control store at one time. He identifies the very difficult problem of "providing system I/O interface facilities that are general enough to support a wide range of emulators", and proposes a low level interface allowing "relatively powerful I/O requests from the emulator". He ·also specifies the following system characteristics:

    1) Transparency: a minimum of system intervention in
the normal operations in the emulator and in the virtual
machine program;
    2) Generality of emulation: a minimum of
restrictions on the structure and operation of the
emulators;
    ?) Size: a minimum of resident system functions
during processing of jobs.
    Briefly stated, the system goal is the ability to
run a sequence of jobs, under arbitrarily many different
emulators, without manual intervention to load
emulators, check timing, and so on.

Hopkins' I/O interface is based on the specification that I/O need not be carried out directly: a method of stacking and processing I/O requests is suggested. He also suggests a set of "primitive" I/O procedures which any emulator can call. He does not, however, address the problem of handling interrupts in any great detail, and even hints that they may never be seen above the lowest level of program control. This is generally unsatisfactory when attempting to achieve something approaching real time response, especially when the emulated

machines themselves have an interrupt structure. The resident
package consists of the currently running emulator, the I/O
interface, an error recovery routine, an "end of task"
routine, and optional accounting routines. A number of
restrictions are outlined, including use of the system I/O
interface routines, protection of control store tables,
register restrictions imposed by the system, and clean-up by
the exiting emulator. He acknowledges that enforcement of
these restrictions requires co-operation between emulator and
system, but states that they do not restrict the structure of
any particular emulation. Research areas listed are: provision
of real-time communication, concurrent residence of more than
one emulator, and inclusion of microdiagnostics (perhaps in
the error recovery routine).

• McDonnell Douglas Astronautics Corporation

A very specialized form of multi-emulation has found
application in the emulation of the MAGIC 352 computer (Delco
Electronics, 1973) by Dalrymple and Durakovich (1974). Theirs
is a dual emulation, "an emulation of the new computer
combined with an emulation of an existing computer, in order
to take advantage of the support software already available
for the existing machine." The host machine is the DSC META 4
(Digital Scientific Corporation, 1972), a microprogrammable
machine with 32-bit read-only memory and two 64 word (16-bit)

scratch-pad memories. The existing computer emulator is an IBM 1130, and the new emulator is the MAGIC 352.

> A master-slave relationship was established between the IBM 1130 and the MAGIC 352 emulations by providing a new 1130 instruction to "turn on" the 352. When it is executed, the 1130 emulation stops and the 352 emulation begins execution. Whenever an exceptional condition occurs, control returns to the 1130 emulation and a flag is set which the 1130 program can use to determine the nature of the condition. Communication between the two emulations is accomplished via the scratch-pad memories.

The system includes commands to modify and dump MAGIC memory, modify the MAGIC clock, and arm and disarm interrupts. Also provided are a checkpoint/restart facility, a breakpoint capability, a tracing capability, and a snap dump facility.

• Nanodata Corporation

Nanodata Corporation's QM-1 computer is a machine specially suited for emulation. Of particular interest are its two levels of microprogram control and large number of independent data buses. The company, as part of its firmware and software support, has provided a multi-emulator environment with its QM-1 computer (Nanodata Corporation, 1974). One of their firmware supervisors, called CONTROL, supports concurrent execution of multiple NOVA 1200 emulators and provides firmware device drivers for a wide range of I/O devices. Much of the basic design of the multi-emulator environment described in Chapter IV came from studying CONTROL. CONTROL is completely vertically microcoded and

resides in an 18-bit control store. The definition of MULTI, which is the microinstruction set in which CONTROL is written, requires approximately 128 36C-bit words of nanostore. The NOVA emulator itself occupies about 6C words of nanostore. No storage swapping (other than register save and restore) is required on emulator switch, which occurs on a clock interrupt. Each emulator accesses its own disk pack, and has access to all other devices. Although many design and implementation considerations are simplified because the environment is homogeneous, CONTROL is a reasonable effort at providing a computer emulator environment.

• IBM Corporation

A paper by Allred (1971) concerns itself with "the design and development of integrated emulators for the IBM System/370." Following are the design criteria used in these emulators as presented by Allred:

1. Emulators must be fully integrated with the operating system and run as a problem program.
2. Complete multiprogramming facilities must be available including multiprogramming of emulators.
3. Device independence, with all device allocation performed by the operating system.
4. Data compatibility with the operating system.
5. A single jobstream environment.
6. A common, modular architecture for improved maintenance and portability.
7. An improved hardware feature design with emulator mode restrictions eliminated and all feature operations interruptible.

Although these emulators fill a real need, i.e. letting a user keep on using software for an old machine while

"growing into" a newer one, they are not multi-emulator
environments in the sense indicated in the first chapter.
First, I/O is simulated by providing an "emulator access
method" to the 370 operating system. Second, because the
emulator is running as a problem program, it may not use
privileged opcodes and hence is not a complete emulation.
Third, although multi-emulation might be supportable, it was
apparently never done. The "integrated emulation" concept is
interesting because the emulator does not run stand-alone; it
must communicate with the 370 operating system. This is
similar to communication between a multi-emulator supervisor
and a member emulator, and could provide some valuable tips to
the multi-emulator system designer.

* Burroughs Corporation

A report by Davis, Zucker, and Campbell (1972) describes
"a structure for connecting and controlling a multiprocessor
system using a building block technique."

> The hardware is modular and includes microprogrammable
> processors called "Interpreters", memory modules, and
> devices. Each Interpreter is interconnected with every
> memory module, and every device via a data exchange
> network called a "Switch Interlock".

> The Interpreters, also known as "D-machines", are identical
> and have a writable control memory. See Reigel, Faber, and
> Fisher (1972) for a detailed description of the Interpreter.

> In the multi-Interpreter System, ... I/O control and
> processing functions are all performed by identical
> Interpreters, and any Interpreter can perform any
> function simply by a reloading of its microprogram

memory. In the Multi-Interpreter Control Program I/O
operations simply become tasks which are
indistinguishable to the control program from data
processing tasks except that they require the possession
of one or two I/O devices before they can begin to run.
(A task is defined as an independent microprogram and
its associated "S" level program and data, which
performs explicit functions for the solution of user
problems.) Whenever an Interpreter is available it
queries the scheduling tables for the highest priority
ready-to-run task, which may be an I/O task, a
processing task, or a task which combines both
processing and I/O functions.

It is claimed that the Interpreters have successfully
performed the following tasks: emulator, peripheral
controller, high level language executor, and special function
operator. This work is of interest because it describes an
operating system which controls concurrently running emulators
(albeit in a multi- rather than single-processor environment),
and because the Interpreters can be made to change their
function. Thus the intent of this system (modular multi-
emulator control) is basically the same as that of the system
to be described herein, although the actual structure is
vastly different.

The Burroughs B1700 computer, introduced by Wilner
(1972a), is a small-scale, bit-addressable, microprogrammable
computer. System design centers around a set of "S-languages"
each of which is suited to the accomplishment of a fairly
specific task, usually interpretation of source code for a
particular high level language.

The B1700's objective, consequently, is to emulate
existing and future S-machines, whether these are 360's,
FORTRAN machines, or whatever. Rather than pretend to be
good at all applications, the B1700 strives only to

interpret arbitrary S-language superbly. The burden of
performing well in particular applications is shifted to
specific S-machines.

The claim is made that the process of the B1700 hardware
interpreting an S-machine, which in turn is interpreting an
application program, "is more efficient than a single system
when more than one application area is considered." The B1700
Master Control Program (MCP) handles all virtual memory
requirements, and all multiprogramming functions (I/O, storage
management, and peripheral assignment). All I/O is done by
communicating descriptors; I/O has its own S-language. MCP
itself is written in a higher-level S-language (SDL) and
interpreted. The system appears to be very flexible in
switching between interpreters:

> Interpreter switching is independent of any execution
> considerations. It may be performed between any two S-
> instructions, even without switching S-instruction
> streams. That is, an S-program may direct its
> interpreter to summon another interpreter for itself.
> This facility is useful for changing between tracing and
> non-tracing interpreters during debugging.

The system is claimed to be very efficient at emulating high
level language machines. Wilner (1972b) includes some very
favourable statistics when describing the COBOL, RPG, and
FORTRAN S-machines on the B1700. As well, S-machines have been
designed and implemented by others; for example, see Belgard
(1974) and Firestone (1973). However, it seems that the B1700
would not be particularly good at emulating sophisticated
general purpose computers, particularly because of the small
number of general purpose registers (four) and the fact that
control store can hold only microinstructions. A MITRE

Corporation report, by Burke, Gasser, and Schiller (1974),
describing a feasibility study of emulating the Honeywell
6180, supports this impression:

> The benchmark emulation of the STA ·instruction ...
> requires 76 microinstructions. Assuming 250 nanoseconds
> ... per microinstruction, this is an execution time of
> 19 microseconds. Delays due to main memory access could
> easily increase this time to 25 microseconds. [The
> execution time of a real 6180 is 1 microsecond.] Several
> other factors diminish the feasibility of emulating a
> 6180 with a B1700. The maximum size of control memory on
> the B1728 is 4096 microinstructions. It is highly
> unlikely that this is large enough for a full emulation
> -   thus either microinstructions would have to be
> executed out of main memory (which the B1700 can do) · or
> micro routines would have to be "demand paged" into
> control memory. In either case performance would suffer.
> Also, the maximum size of main memory - 256K bytes for a
> B1728 - is very small compared to a "typical" 6180
> configuration. Even the theoretical limit of 2^{21} bytes
> of main memory (based on the size of the memory address
> register) is on the small side.

This same report estimates a time degradation of a factor of
six to ten using the QM-1 rather than the B1700 as the host
for a 6180 emulation.

CHAPTER III

Multi-Emulator Supervisor Design

3.1 Introduction

This chapter deals with the design of a multi-emulator
supervisor. The following section states the basic functions
of such a supervisor. Next, the general design principles
embodied in a multi-emulator supervisor are presented. It
should be noted that the principles to be outlined are
particular to a multi-emulator environment; principles drawn
from operating systems theory will not explicitly be stated.
These principles are then expanded into a compact set of
fairly explicit guidelines useful to the designer of a multi-
emulator system. Finally, the possibility of providing a high
level I/O interface to members of a multi-emulator environment
is examined.

3.2 Supervisor functions

A multi-emulator supervisor must have several basic
functions in order to provide the necessary support for and
exert the necessary control over member emulators. These
functions, including a task scheduling and switch capability,
operator supervisory and control functions, and a low level
I/O interface, are described briefly in this section. A more

detailed discussion of these matters is presented in the next
chapter, as the functions are better understood in the light
of an actual implementation. Figure 2.1 shows the logical
position of the supervisor in a multi-emulator system.

A task switch mechanism must be provided, its precise
form depending upon the design philosophy. For example, in a
batch mode of operation as suggested by Hopkins (1970), the
incoming emulator's state may be assumed to be uninitialized
(corresponding to a machine's state immediately after power-
up). However, in a multi-tasking environment, emulator states
must be preserved across task switches, and the mechanism
becomes more complex. Storage swapping may be required. Also,
as we shall see, the problems of device ownership and time-
sensitive operations become thorny indeed. For flexibility, a
number of types of events should have the potential to trigger
task switch; these should include timer interrupt, I/O device
interrupt, instruction step, and perhaps explicit emulator
request. Also for flexibility, the event types which actually
are allowed at any one time to cause a switch should be
operator-selectable.

The control program must have some means of interacting
with the system operator. Commands for storage display and
modification are necessary, as are commands to start and stop
member emulators. The operator must have at least as much
control over a member emulator as he would have if he were
operating the real computer from its front panel. Experience

```
  ┌ ─ ─ ┐        ┌ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┐
  │     │   ┌── │   supervisory      │              ┌ ─ ─ ┐
  │     │   │    │   and control     │              │     │
  │     ├── ─ ─ ─│                   │          ┌───│     │
  └ ─ ─ ┘        │                   │          │    └ ─ ─ ┘
                 │   ├── ─ ─ ─ ─ ─ ─ ─│── ─ ┘
     .           ├── │ device drivers │── ─┐        .
     .           │    │                │    │         .
     .           │    ├── ─ ─ ─ ─ ─ ─ ─│    │        .
  ┌ ─ ─ ┐        │    │                │    └ ─ ─┐
  │     ├── ─ ─ ─│    │   task switch  │          │   ┌ ─ ─ ┐
  │     │   └── │                     │          └──│     │
  │     │        │                   │              │     │
  └ ─ ─ ┘        └ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┘              └ ─ ─ ┘
```

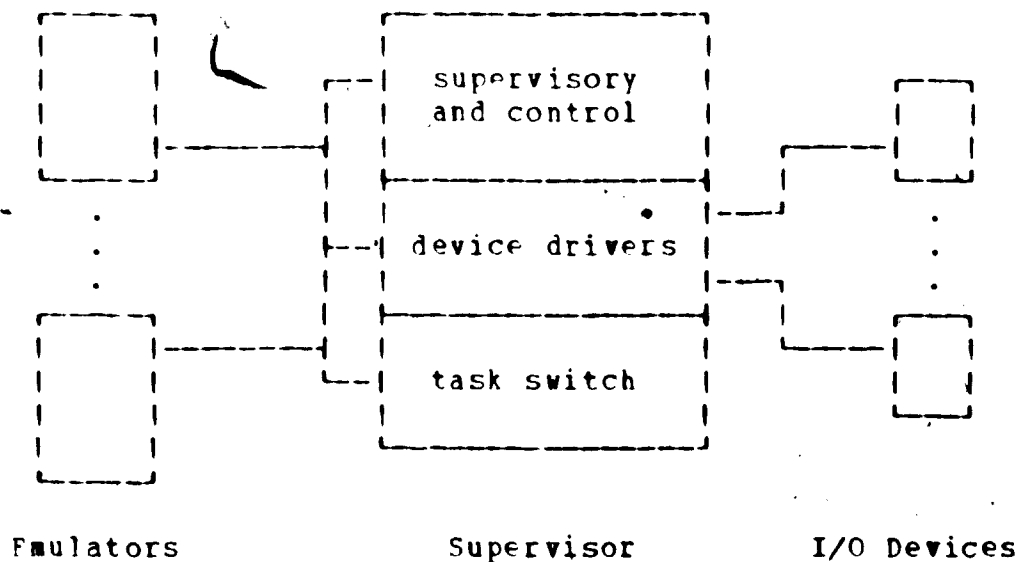Emulators   Supervisor   I/O Devices

Fig. 3.1. Basic Multi-Emulator System Structure

shows that debug facilities such as register tracing and single step mode are very desirable. Provision should be made for the inclusion of automatic system monitoring facilities. The operator should also be allowed to dynamically modify an emulator's configuration. (e.g. add a tape drive, reassign the card reader).

Since the system will include emulators capable of carrying out I/O operations at a very low level, it must provide support for such operations. A standard set of firmware device drivers and a bi-level interrupt mechanism are necessary. These important concepts will be explained and their inclusion justified in the following section.

3.3 Design Principles

In order to better grasp the concepts of multi-emulation, consideration of the principles involved is warranted. This section presents two underlying principles of multi-emulation. It should be stressed again that it is computers being emulated, not simply processes.

3.3.1 Architectural Interference

A multi-emulator environment will in general consist of emulators with very different machine architectures. If the supervisor imposes some basic architectural constraints that do not already exist in the host machine's architecture, then the varieties of emulation possible may be greatly reduced, and emulator design to fit these additional constraints may become more difficult. One might argue that providing complete freedom to member emulators is asking for trouble with respect to protecting one emulator from another. This is true, but it is more important to minimize interference and assume that the emulators are well-behaved, than to restrict severely an emulator's access to the machine for the sake of protection. After all, a multi-emulator environment is a very special one, not containing users out to "break" the system. Something the supervisor can do is provide safe facilities for the member emulators to use, especially in terms of I/O.

3.3.2 Exact Emulation

The supervisor (and the host machine) should have
facilities to provide _exact_ emulation of computers. This
principle is visible primarily in the consideration of
communication between an emulator and the real world, i.e.
emulator/device interfacing. Emulator/device interfacing is
the single most important consideration to the multi-emulator
supervisor designer.

Here is a summary of some of the specifications
presented earlier in the light of providing exact emulation.
First, the supervisor will allow emulators to communicate
directly with I/O devices. When device numbers are limited, as
is often the case, devices must be shared among emulators:
this service is the responsibility of the supervisor, meaning
the provision of both standard device/emulator interfaces and
dynamically alterable device ownership. Second, the member
emulators will be computers, and most computers deal with I/O
devices at a low level; therefore, the supervisor must provide
a suitably low level interface to devices. Thus, the
supervisor must combine a standard device interface with a low
level device interface. The following subsections consider
these points more closely.

3.2.2.1 Device Ownership

Clearly, more than one type of device ownership must be supported. The types shall be categorized as follows:

* absolute -

  This term is applied to devices which are owned solely by a single emulator. For example, a computer's console would probably be owned absolutely. This type of ownership is easiest to support, requiring no special supervisor support to route interrupt signals, status information, and the like.

* sequential -

  This applies to devices which need a long term ownership but which must be shared among emulators. Examples are line printers, magnetic tape drives, card readers, etc. Supervisor support for this type of ownership becomes much more complex than for absolute ownership because a number of new problems arise.

  First, a mechanism is required to cause a change of ownership from one emulator to another. If the I/O were done at a high level (i.e. the open, close, read, write level of a multiprogramming operating system), then the problem would not be difficult: when an emulator opened a device, it would be given ownership of it until it closed the device. (Of course, other problems such as what to do when a device is

required by an emulator while in use by another, and what to
do about an emulator which "forgets" to close a device would
have to be overcome.) However, since the members are
computers, and I/O is carried out at a low level, the
supervisor cannot in general know when an emulator has
finished using a device; the virtual machine has no "close"
instruction to indicate this. For example, consider the
listing of a file on a line printer: how is the supervisor
to know that the emulator is has completed the listing,
unless the emulator has some means of explicitly indicating
this to the supervisor? The following simple sequential
device switch mechanism is therefore proposed: an operator
command would be provided to allocate a device to a given
emulator, and the device would be made to appear not ready
to any other emulator. Of course, the operator should
satisfy himself that the device is not in use before
reallocating it.

A second major problem not encountered in the absolute
ownership case is the provision of a standard firmware
device interface which can communicate with any emulator.
This problem deserves a section of its own (section
3.3.2.2).

Manual sequential ownership as described above may be
unsuitable in some real time applications, for example a
system of process control emulators attempting to share data
acquisition devices. For one thing, the delays caused by

supervisor overhead and by the fact of emulation itself.
might be such that the emulator could not handle interrupts
quickly enough, or that supervisor-imposed changes in
interrupt timing might render results invalid. For another,
the act of manually switching device ownership would almost
surely result in lost interrupts. (Such a system, however,
would probably be usable in a batch rather than concurrent
mode of operation.)

• shared -

Sequential ownership, with its clumsy method of device
ownership allocation, is far too slow for some devices,
notably disk drives. Especially in small configurations,
different emulators may require the services of the same
disk controller in rapid succession; a type ownership of a
very temporary nature is therefore required.

Here is a possible solution to this ownership problem.
For each disk drive, the supervisor would remember the most
recent cylinder address sought to by each emulator sharing
the disk. The currently running emulator would automatically
become disk owner, and any disk access made by it would be
delayed until the supervisor had done a "safety seek" to
ensure the heads were properly positioned. Of course, the
supervisor would have the responsibility of protecting other
emulators' data; the VM/370 "mini-disk" concept (IBM
Corporation, 1972) is useful here. A related problem is the

possibility that task switch might occur while data is being
transferred to or from disk. To solve this problem, the
supervisor could set a lock upon commencement of data
transfer and reset the lock when the transfer is complete;
while the lock was set, task switch would not be permitted
(see section 4.?.?).

?.?.2.2 Standard Device Drivers

Each device type requires a standard firmware device
handler which can communicate with any emulator, especially if
the device may be shared among member emulators. This
interface is necessary for the routing of signals to and from
the current device owner and to guarantee that only the
current owner may access the device. Naturally, as much as
possible of the I/O processing should be done by the
interface, but unfortunately interfacing cannot be provided at
a very high level because (once again) of the average
computer's ability to carry out low level I/O. For example,
standard multiprogramming features such as device-independent
I/O and spooling of data are simply out of the question.
Provision of standard device drivers is an illustration of an
important principle of multi-emulation: the supervisor should
be interposed between device and emulator, without exception.
Another illustration of this principle follows.

3.3.2.3 Interrupt Handling

The supervisor must include an interrupt handling routine to take interrupts directly from the I/O devices. Thus, when an interrupt occurs, the supervisor will remember it in its own queue, and will signal the appropriate member emulator. The member emulator will in turn remember the interrupt in its own fashion, and pass control back to the supervisor to dequeue the device from the supervisor's interrupt queue. This method, called bi-level interrupt handling, has two very favourable features. First, no reliance whatever is placed on a member's interrupt handling mechanism. In fact, it need not even be resident when an interrupt occurs. Second, each member's interrupt mechanism can remember the interrupt in any way it sees fit. This could range from a priority interrupt scheme (as in a PDP-11) to simply setting a flag and forcing control in the emulator to pass to a standard entry point (as in a PDP-8). Maximum flexibility is therefore allowed.

3.4 Specific Guidelines

Basic design principles and their ramifications having been considered in the previous section, these criteria may now be reformulated into a set of guidelines for the multi-emulator supervisor designer.

• The supervisor must impose minimal architectural constraints

upon member emulators. See section 3.3.1.

- If possible, the microinstruction set used to implement the supervisor should be tailored to maximize its efficient coding. This is obvious, and at the same time very difficult for most user-microprogrammable machines.

- Actions by the supervisor not directly affecting the active emulator's storage must be invisible to it, as must actions carried out by other member emulators. In particular, task switch must be unnoticed by an emulator. This guideline is easy to follow if constant attention is paid to it during design; it is nonetheless important, as it doesn't permit various sorts of painful interdependencies to build up. Of course, supervisor functions may exist to explicitly affect an emulator, for example in supervisor-controlled inter-emulator communication.

- Debug and supervisory capabilities must be provided the operator by the supervisor, especially for the emulator designer. These were discussed in section 3.2.

- There should be a number of types of task switching available to the operator. Also, a lock facility must be provided to delay task switch during critical operations. The business of task switching was also discussed in section 3.3. Provision of many types of triggers is beneficial

especially in environments where it is not known which types
would provide the smoothest running system. Much work can be
done concerning task switching, perhaps along the lines of
similar multiprogramming concepts such as swapping and
paging. This matter will not be examined further in this
thesis.

* The supervisor must be interposed between emulators and the
physical devices themselves, and, to the greatest extent
possible, code should be shared between different emulators'
firmware device drivers for the same device type.
Interposition is an important concept because it aids both
in device sharing and in reducing the chance of possible
inter-emulator interference.

* A bi-level interrupt control mechanism must be provided by
the supervisor: interrupts will be handled by the supervisor
first, and then allowed to activate the owner's interrupt
mechanism. This may be thought of as a specific application
of interposition, but it is important enough to warrant
special note.

* More than one type of device ownership must be provided. The
categorization presented in section 3.3.2.1 included
absolute, sequential, and shared ownership.

The following chapter seeks to apply these guidelines to a real multi-emulator system. Emulator/device interfacing forms a major part of the discussion, either directly or indirectly.

?.5 Note on a High Level I/O Interface

Although the provision by the supervisor of low level, direct communication between emulator and device is a more flexible approach than any other, it has its drawbacks:

- Any new emulator type being added to the system must have a complete set of device drivers written for it, at best using existing drivers as models. This is a time consuming process.

- While a low level interface is extremely suitable for computer emulators, it is extremely unsuitable for language emulators.

A worthwhile effort would be the provision in a multi-e▮▮▮▮▮onment of a high level I/O interface. The reader s▮▮▮▮▮ the references provided in section 2.3 if he is in▮▮▮▮n this area. Here are some criteria for such an int▮▮

▮▮▮ust not impose architectural constraints upon its ▮s.

▮ I/O should be device independent.

▮▮▮ concept of "opening" and "closing" devices should ▮ employed in order that there be no device ownership ▮roblems.

As this thesis is concerned mainly with supporting traditional computer emulators which require a low level

interface, this section is not meant to be a solution to the problem of providing high level I/O at the microprogram level, but merely an outline of some of its aspects. Many capabilities must be provided: a file system, spooling facilities, the ability to handle various data formats, and the handling and communication of error conditions, to name a few. Even with all these problems, however, the elevation of the I/O interface to a high level should make design and implementation of a system comprised of processes which use the interface a much simpler task than that of providing an environment for exact computer emulation. Opportunities for both theoretical and practical research exist in abundance.

In addition to providing high level I/O for the purpose of supporting language processors, another interesting direction for research would be the investigation of computer architectures with an eye to improving the I/O structure. Traditional computer order codes are unbalanced: accomplishment of I/O functions is an order of magnitude more complex than accomplishment of any other function. The programmer must get "closer to the machine" than at any other time; he must do more bit manipulation and usually has timing problems to worry about. The problems are akin to those which we would have to suffer through if our ADD and SUBTRACT instructions were replaced by an EXCLUSIVE-OR capability in which propagation delay had to be considered. It is no surprise at all that operating systems hide low level I/O

activities from the average user: they are simply too much of a headache! Work toward the provision of device-independent, high level I/O instructions for a contemporary computer's main store instruction set, performing all the necessary low level activities at the microprogram level, would certainly be worthwhile.

CHAPTER IV

A Particular Multi-Emulator Environment

4.1 Introduction

Setting down a list of guidelines for the multi-emulator designer to follow is all very well, but what of showing that they are actually workable? This chapter gives a brief description of a multi-emulator system in which the guidelines suggested are applied. The following section is an overview of the system: setting, member emulators, and major data structures. The system's logical structure comprises the final section. Illustrations of how the guidelines have been applied are interspersed throughout the chapter.

4.2 Basic Structure

In order to better understand the following sections, the reader may wish to gain some familiarity with the architecture of the host machine for this system; the QM-1. Other than the Hardware Level User's Manual (Nanodata Corporation, 1974), a number of articles present short descriptions of the QM-1. These include: Rosin, Frieder, and Eckhouse (1972); Lutz and Manthey (1972); Dorin (1972); Thomas (1974); Petzold, Richter, and Rohrs (1974); and Agrawala and Rauscher (1974).

4.2.1 Setting

The multi-emulator system to be described (often
referred to in this chapter as the "system" and the
"supervisor") is for the Nanodata QM-1, closely modelled after
their system called CONTROL (see section 2.3). The supervisor
is written in MULTI, an assembly-like vertical
microinstruction set defined by Nanodata Corporation. A
favourable feature of MULTI is that unlike most
microinstruction sets, timing problems do not occur: all
actions such as arithmetic operations and memory accesses are
synchronized. Thus, although no parallelism can be achieved,
code is relatively easy to write, understand, and debug.
Execution times of MULTI instructions averages approximately
.5 microseconds per instruction. The supervisor resides
completely in control store; that is, the control store holds
both the code and the data structures which make up the
system. In the implementation each member is a computer
emulator, so the system's basic functions are these: to
provide low level I/O support to member emulators; to provide
a task switch capability; and to provide the system operator
with control and supervisory functions.

It should be noted that although all of the components
of the system have been written and tested, they have not been
assembled into a complete system. This was due in part to a
lack of availability of certain peripheral devices, and also

to the lack of stability in the QM-1 hardware caused by break-
in troubles and by shipping damage.

4.2.2 The Member Emulators

As has been mentioned, the member emulators making up
the system are computer emulators. One is a Nova 1200 emulator
written by Nanodata Corporation; the other is a PDP-11/10
emulator (Demco and Marsland, 1975). Information regarding the
architectures and instruction sets of these computers may be
found in reference manuals (Data General Corporation, 1972;
Digital Equipment Corporation, 1973).

Neither emulator was written with concurrent operation
in mind. This points out the generality of the design
principles, especially that of minimization of architectural
interference. An emulator can, and in fact should, be designed
for stand-alone operation, and the supervisor will allow for
its inclusion in the system without modification to the
emulator or to any other member emulator.

The emulators' structures are similar. They are not
written in a vertical microinstruction set which is in turn
interpreted by nanocode; instead, main store instructions are
interpreted directly by nanocode. Also, all I/O operations are
handled by firmware routines which reside in control store,
and are coded in the vertical microinstruction set MULTI. The
emulators are reentrant. Both have a similar instruction fetch

●

and decode routine:

- The program counter is used as pointer into main store, from which the next instruction is fetched.

- A flag is tested. If it is set, the emulator passes control .to a microroutine instead of proceeding with the instruction execution. The act of diverting the emulator to this routine is termed a logical_interrupt. The reason for having this flag is explained below.

- The program counter is updated.

- Some number of bits of the instruction are used as an index into a table in control store from which is taken the address of the appropriate nanoroutine to execute the instruction. The size of this table for each emulator is 512 words.

- The nanoroutine is executed, performing the required function. Preliminary actions may occur, depending on instruction type (e.g. effective address calculation). Also, side effects may occur, such as auto-incrementing of a register or memory location.

- If the instruction performs I/O, control passes to a microroutine in control store to handle the request; else control passes back to the beginning of the instruction fetch routine to execute the next main store instruction.

The Nova instruction set is more suited to this sort of decoding technique, as a quick look at the formats of the two instruction sets will show. The emulators are started by special microinstructions added to MULTI: these microinstructions simply cause control in the QM-1 to pass to a particular nanoaddress (the start of the emulator's fetch and decode routine) rather than continuing with the execution of the next microinstruction.

The logical interrupt flag is usually set by an interrupt handler, indicating that one of the emulator's

devices requires service. If this is the case, the microroutine invoked causes the emulator to recognize the interrupt in whatever fashion the emulator's architecture demands. For example, the Nova's routine merely forces a subroutine call to main store location 0. The logical interrupt flag may also be set by the command handler (this is explained in section 4.3.4).

The Nova emulator runs at between .3 and 1.25 times the speed of a real Nova 1200, and the PDP-11 emulator runs at .5 to 1 times the speed of its target machine. Unfortunately, actual computers were not readily available for comparison of execution times of sample programs. However, a benchmark standalone BASIC program took about 4.4 times as long to execute on the PDP-11 emulator as it did on a real PDP-11/45, a machine considerably more powerful than the PDP-11/10.

The control store on the QM-1 acting as the host machine for the system is large enough (5K 18-bit words) to hold the supervisor and device drivers for both emulators at once. The supervisor requires approximately 1600 words for task switching, command handling, and task control blocks (see section 4.2.3.1). The Nova, which supports a full complement of I/O devices (teletype, card reader, line printer, moving head disk drive, and cartridge tape unit), occupies about 1500 words. The PDP-11 presently supports only a teletype and a high speed paper reader, and requires approximately 500 words. (Incidentally, the paper tape reader device driver actually

reads cards; this was done because of the ready availability
of paper tape software.) Total control store utilization,
including 1024 words for emulator instruction decode tables,
is approximately 4600 words. The nanocode for the definition
of MULTI and for both emulators also may reside in nanostore
concurrently: MULTI occupies about 125 360-bit words, the Nova
about 60, and the PDP-11 about 125. The host has 512
nanowords. In addition, main store is large enough for one
copy of each emulator's main store: each emulator can address
32K words of main store, and the host presently has 64K words.
(The QM-1 is equipped with a simple option for mapping main
store addresses which permits more than one emulator's address
space to be resident concurrently. However, the option is not
sophisticated enough to provide an efficient paging
mechanism.) Thus, for a simple system comprised of one
emulator of each type, no storage swapping (other than
registers) is required when task switch occurs.

4.2.3 Data Structures

The supervisor manages two major data structures in
controlling devices and member emulators. These data
structures reside in control store, as fast access is
required. The following is a description of each.

4.2.3.1 Task Control Block

For each member emulator, there is one TCB containing
data used to control and determine the state of the member. It
is comprised of both emulator-independent and emulator-
dependent information. The emulator-independent information
includes a save area for local store, external store, and P-
store; a logical halt indicator (set when the emulator is
halted or inactive); a single step indicator; a word
indicating which console command, if any, is pending for the
emulator; a word indicating how many more clock interrupts
must occur before this task will be deactivated; and a
constant to which the clock counter is set when the task is
activated. For a PDP-11 emulator, the dependent information is
a copy of the switch register, a flag to indicate whether or
not this emulator is executing a WAIT instruction, and a
pointer to a priority-ordered linked list of unit control
blocks which the emulator owns and which are demanding
interrupt servicing. The Nova's dependent information consists
of its logical sense switches and a flag to indicate whether
or not it may be interrupted by its devices. Each TCB requires
approximately 50 control store words.

4.2.3.2 Unit_Control_Block

The UCB is used by the supervisor to control the device
it represents. Device independent information is comprised of
a link field for queuing the UCB on the lower level interrupt
queue, an entry point address into the upper level interrupt
handler, the current owner indentification, the physical
device address, and status information. The device-dependent
information for devices currently belonging to PDP-11
emulators includes a link field for the emulator's priority
interrupt queue, the trap vector address, the bus request
priority, and copies of the device registers. Nova device
information is a list of handler entry point addresses
(corresponding to input, output, test, and null I/O
operations), a done flag, and a busy/inactive flag. UCBs
occupy approximately 15 control store words each.

4.3 Logical_Structure

Each of the major components of the multi-emulator
system is described in this section. The components are: task
switch, I/O initiation, interrupt processing, and command
handling.

4.?.1 Task Switch

Because in the implementation of this system the storage requirements for both emulators do not exceed that available on the QM-1, task switch is easy to accomplish. It is triggered by timer interrupt only, although other switch signals (see section 3.2) may easily be added at some later time.

Upon receipt of a timer interrupt, the task lock is tested. If set, the lock indicates that a critical operation is under way (e.g. DMA transfer) and task switch must be delayed until the active emulator resets it. If the lock is not set, the supervisor saves the currently active emulator's register contents in the designated TCB save area, chooses the next emulator to become active (a simple rotation suffices here), restores its registers from its TCB, and passes it control. These actions require approximately 20 microseconds in total.

4.3.2 I/O Initiation

When an emulator decides to perform an I/O operation, a microroutine is called to accomplish the function. There are a number of reasons for handling most I/O requests in microcode rather than directly in nanocode. Most importantly, the amount of nanostore is limited to 1024 words, and all of the device drivers necessary would almost certainly not fit commonly.

Also, the writing of efficient nanocode is difficult and
requires experience, demanding more time than should be spent
on anything except very critical system components.

Before the I/O operation proceeds any further, the
supervisor interposes itself, ensuring that the emulator is
current owner of the device it is attempting to access (by
checking the owner field in the device's UCB). If the
identification matches, the emulator may proceed, having full
access to the device. (Disk access is an exception if mini-
disking is employed. See section 3.3.2.1.) If not, the device
is simply made to appear not ready to the emulator. The
unexpected appearance of an unready device could conceivably
cause problems in some operating systems, so the operator
should exercise care when reassigning devices. In this
implementation, this is the extent of the "standard device
driver" concept explained in section 3.3.2.1. The
identification check is simple, creating very little overhead
yet giving emulators direct access to devices in order to
carry out low level operations with efficiency.

4.3.3 Interrupt Processing

The most important feature of the system's interrupt
structure is its bi-level nature (see section 3.3.2.3). When
an I/O interrupt occurs, the supervisor gains control. A small
number of local store registers and P-registers are saved. The
lowest level interrupt routine places the interrupting

device's UCB on a queue (MULTI has a special microinstruction,
ENQUE, for this purpose), determines the device's status, and
invokes the upper level interrupt handler for the device
(using the entry point address found in the UCB). In this
implementation, this routine is guaranteed to be resident. The
upper level routine determines which emulator is device owner
and passes control to the appropriate subroutine, in which the
interrupt is recorded in whatever manner the emulator chooses.
For example, the PDP-11 emulator device driver does whatever
action is immediately required (usually merely getting
detailed status information from the interrupting device),
places the UCB on its own priority-ordered interrupt queue,
and sets a flag to logically interrupt main store processing
(see section 4.2.2). The subroutine then returns control to
the supervisor which dequeues the UCB from the lower level
queue (a DEQUE microinstruction is provided), restores the
small number of local store registers and r-registers, and
resumes the interrupted task.

Interrupts are disabled in supervisor code only during
queuing and dequeuing of UCBs; in emulator code they are
disabled whenever the emulator chooses, usually during its own
queuing process. Interrupts are stacked at the lowest level;
however, a second interrupt from a device before the first
interrupt has been handled and its UCB dequeued results in the
second interrupt being lost. The lowest level interrupt
handler usually requires less than 7 microseconds from the

moment the QM-1 CPU takes the interrupt until the moment the upper level interrupt handler is entered; typical total handling   from interrupt until the restart of the act emulator   order of 25 to 30 microseconds for a device belongin   emulator.

4.3   dling

          ned in section 3.2, the multi-emulator system shoul   the operator supervisory, control, and debug capabi   . Commands are provided to display and modify an emulato   torage (main store, control store, and registers), to set   ear single step mode, to cause selected registers to be tr   to initialize and reassign devices, and to start and stop   r emulators.

The   anism used to effect an operator command is straightf   hen the interrupt handler for the operator's console   izes a command for a particular member emulator, it records the command in that member's TCB and signals it with a logical interrupt. The next time the emulator begins execution of a main store instruction, it will test its logical interrupt flag and invoke its logical interrupt microroutine (section 4.2.2), which in turn will invoke the command handler. The command handler will execute and return control to the logical interrupt routine, which will restart the emulator. In this way, commands issued by the operator are invisible to the running emulator except for timing

considerations, unless of course the command was intended to
affect it.

In a system with a small hardware configuration, it may
be necessary to have one of the emulator consoles double as
the   system operator's console. This is easily done by
providing some simple mechanism to "point" keyboard interrupts
at the appropriate handler, either the emulator's or the
system's. Similarly, more than one emulator may use the same
console device.

CHAPTER V

Conclusion

5.1 Results

The purpose of this thesis was to consider aspects of multiple concurrent computer emulation. As was expected, experience in the design and implementation of a real multi-emulator system has shown that most problems encountered are associated with providing an environment for exact computer emulation. The task switch mechanism is straightforward, as are the control functions. Minimization of architectural interference by the control program turned out to be a simple matter for well-behaved emulators. The interposition of the firmware support system between each emulator and each device has proved to be both practical and beneficial. However, emulator/device interfacing is a serious problem. Due to the ability of a computer to carry out I/O at a very low level, the envisioned standard interface between a particular device and any emulator occurs at a much lower level than originally expected.

The complexion of the device/emulator interface changes drastically when the emulator is a high level language machine, reflecting the much higher level of I/O. The implementation of a multi-emulator system wherein all member

emulators perform high level I/O exclusively is expected to be
a much simpler task than that of providing an environment for
exact computer emulation. In fact, I believe that in the
(perhaps distant) future only device-independent I/O will be
available to the main store programmer of a large general-
purpose computer. This is an intriguing prosp███

The principles and the guidelines which have been
developed are sound. Their applicability has been demonstrated
in the design of a real multi-emulator system, and I expect
that results will be equally favourable in implementations on
other user-microprogrammable computers.

5.2 Suggestions for Further Research

Design and implementation of a mechanism for providing
machine hierarchies (section 2.2) would certainly prove
fruitful. In particular, the design of a general purpose,
microprogrammed, symbolic debugger would be a great asset to
any multi-emulator system, especially if the member emulators
had been designed to indicate the complete state of their
environments to the debugger. Furthermore, designing an
emulator with the question "How can I make this easily run
under a symbolic debugger?" is sure to make for more
structured design, and hence lead to better structure in user
programs.

The idea of placing low level I/O functions in the

firmware and leaving the main store order code with only fairly high level I/O capabilities has only been touched on in this thesis (section 2.5). It is certainly worthy of further research.

Very little attention was given herein to task switching. True, mere accomplishment is simple enough, but providing efficiency is something else again. Several aspects could be investigated, applying results from operating systems theory:

- the examination of task scheduling, which types of events should trigger task switch, and how the operator could "tune" the system; and

- the possibility of partitioning storage and of employing paging hardware for main store and control store, and the application of various paging algorithms.

Protection in a multi-emulator system is a valid research topic. How can one compromise between providing protection and minimizing interference? Can the supervisor make it easier for the member emulators to be well-behaved? As in the previous suggestion, ideas may be taken from operating systems theory, where protection is a major concern.

These suggestions indicate only a small number of the possible avenues of research, and were provided only to spur further work in multi-emulation. I hope they will.

Agrawala, A. and Rauscher, T. (1974). "Microprogramming: Perspective and Status", IEEE Transactions on Computers, vol. C-23, no. 8, pp. 817-837, August.

Allred, G. (1971). "System/370 integrated emulation under OS and DOS", AFIPS Conference Proceedings (SJCC), vol. 38, pp. 163-168.

Belgard, R. (1974). An Implementation of Blaise on the Burroughs B1726, S.U.N.Y./Buffalo Technical Report 81, Buffalo, New York, June.

Burke, C., Gasser, M., and Schiller, W. (1974). Emulating a Honeywell 6180 Computer System, MITRE Corporation, Report RADC-TR-74-137, Bedford, Mass.

Dalrymple, S. (1972). A Microprogrammed, Virtual Associative Memory, McDonnell Douglas Astronautics Company-West, Report MDAC WD 1895, Huntington Beach, Calif.

Dalrymple, S. and Durakovich, G. (1974). The MAGIC 352 Computer Emulation System, McDonnell Douglas Astronautics Company, Report MDC G4972, Huntington Beach, Calif.

Dasgupta, S. (1974). A High-Level Microprogramming Language, M.Sc. Thesis, Dept. of Computing Science, University of Alberta, Edmonton, Alberta.

Data General Corporation (1972). How to Use the Nova Computers, Southboro, Mass., October.

Davis, R., Zucker, S., and Campbell, C. (1972). "A building block approach to multiprocessing", AFIPS Conference Proceedings (SJCC), vol. 40, pp. 685-703.

Delco Electronics Division (1973). MAGIC 352 Computer Programming Manual, General Motors Corporation, Report TR73-125, December.

Demco, J., and Marsland, T. (1975). A Complete PDP-11 Emulator, Technical Report 75-12, Dept. of Computing Science, University of Alberta, Edmonton, Alberta, August.

Digital Equipment Corporation (1973). PDP11/45 Processor Handbook, Maynard, Mass.

Digital Scientific Corporation (1972). Digital Scientific Corporation META 4 Computer System Microprogramming Reference Manual, Publ. 7C43MO, San Diego, Calif., June.

Dorin, R. (1972). A Viable Host Machine for Research in Emulation, S.U.N.Y./Buffalo Department Report 30-72-MU, Buffalo, New York.

Eckhouse,R., (1971). "A high level microprogramming language (MPL)", AFIPS Conference Proceedings (SJCC), vol. 38, pp. 169-1177.

Firestone, W. (1973). Flubbing on the Burroughs B1700, S.U.N.Y./Buffalo Technical Report 64, Buffalo, New York, July.

Hassitt, A., Lageschulte, J., and Lyon, L. (1973). "Implementation of a high level language machine", Comm. ACM, vol. 16, no. 4, pp. 199-212, April.

Hopkins, W. (1970). "A Multi-Emulator Operating System for a Microprogrammable Computer", Third Workshop on Microprogramming, Buffalo, New York, October, (Preprints).

Huberman, B. (1970). Principles of operation of the Venus microprogram, MITRE Corporation, Report MTR 1843, Bedford, Mass., July.

Husson, S. (1970). Microprogramming: Principles and Practices, Prentice-Hall Inc., Englewood Cliffs, N.J.

IBM Corporation (1972). IBM Virtual Machine Facility/370: Introduction, Form GC20-1800-0, Burlington, Mass., July.

Liskov, B. (1972). "The Design of the Venus Operating System", Comm. ACM, vol. 15, no. 3, pp. 144-149, March.

Lloyd, G., and Van Dam, A. (1974). "Design considerations for microprogramming languages", AFIPS Conference Proceedings, vol. 43, pp. 537-543.

Lutz, M. and Manthey, M. (1972). "A Microprogrammed Implementation of a Block Structured Architecture", Fifth Workshop on Microprogramming, Urbana, Illinois, pp. 28-41, September, (Preprints).

Madnick, S. and Donovan, J. (1974). Operating Systems, McGraw-Hill Book Company, New York, New York.

Melbourne, A. and Pugmire, J. (1965). "A small computer for the direct processing of FORTRAN statements", Computer Journal, vol. 8, pp. 24-27.

Nanodata Corporation (1974). QM-1 Hardware Level User's Manual, Second Edition, Williamsville, New York, August.

Petzold, R., Richter, L., and Rohrs, H. (1974). "A Two-Level Microprogram Simulator", Seventh Workshop on Microprogramming, Palo Alto, Calif., pp. 41-47, (Preprints).

Ramamoorthy, C. and Tsuchiya, M. (1974). "A High-Level Language for Horizontal Microprogramming", IEEE Transactions on Computers, vol. C-23, no. 8, pp. 791-801, August.

Reigel, F., Faber, U., and Fisher, D. (1972). "The interpreter - A microprogrammable building block system", AFIPS Conference Proceedings (SJCC), vol. 40, pp. 705-723.

Rosin, R. (1969). "Contemporary Concepts of Microprogramming and Emulation", Computing Surveys, vol. 1, no. 4, pp. 197-212.

Rosin, R., Frieder, G., and Eckhouse, R. (1972). "An Environment for Research in Microprogramming and Emulation", Comm. ACM, vol. 15, no. 8, pp. 748-760, August.

Thomas, R. (1974). "The Development of User Microprogramming: A Survey and Status Report", Seventh Workshop on Microprogramming, Palo Alto, Calif., pp. 212-216, (Preprints)

Tucker, S. (1965). "Emulation of Large Systems", Comm. ACM, vol. 8, no. 12, pp. 753-761, December.

References

Weber, H. (1967). "A Microprogrammed Implementation of EULER on IBM 360/30", Comm. ACM, vol. 10, no. 10, pp. 549-558, October.

Werkheiser, A. (1970). "Microprogrammed Operating Systems", Third Workshop on Microprogramming, Buffalo, New York, October, (Preprints).

Wilner, W. (1972a). "Design of the Burroughs B1700", AFIPS Conference Proceedings (FJCC), vol. 41, Part 1, pp. 489-497.

Wilner, W. (1972b). "Burroughs B1700 memory utilization", AFIPS Conference Proceedings (FJCC), vol. 41, Part 1, pp. 579-586.