PRICM
Informal Comments
Rcvd 4/1

Honeywell

PROJECT GUARDIAN
TECHNICAL COORDINATION LETTER

Date: 30 January 1976

TCL No: 15

Contract No: F19628-74-C-0193

TO: Contracting Officer
HQ/ESD/MCP
Hanscom AFB
Bedford, Mass. 01731

Attention: C. E. Fenton, Captain, USAF

Subject: Prototype Secure Multics - External I/O Functional Description

The attached technical note describes a preliminary study of I/O services in a prototype Secure Multics System. Work in this area will continue during the next phase of the Guardian Project.

If there are any questions, please contact the undersigned or Mr. N. Adleman at our Cambridge, Massachusetts office.

Very truly yours,

HONEYWELL INFORMATION SYSTEMS, INC.

R L Carlson

R. L. Carlson
Contract Specialist

Attachment

cc: ESD/MCI (5)
MITRE/D73, Mr. E. Burke (5)
RADC/ISM (3)
NSA/R14 (3)
AFDSC/XMS (2)
JTSA (5)

PROJECT GUARDIAN

PROTOTYPE SECURE MULTICS
EXTERNAL I/O FUNCTIONAL DESCRIPTION

- Must consider networks
- Positions not supported
  need for IOM
  securability of IOM
  model to design correspondence
  not alluded to
- ~~1.~~ Vague & Incomplete
- ~~2.~~ HIS needs standard
  spec. notation

Technical Note
Preliminary Draft

January 31, 1976

prepared for

Department of the Air Force
Electronic Systems Division
Hanscom Air Force Base
Bedford, Massachusetts 01731

Contract No. F19628-74-C-0193

Honeywell Information Systems, Inc.
Federal Systems Operations
7900 Westpark Drive
McLean, Virginia 22101

Contents

1. INTRODUCTION TO SECURE MULTICS EXTERNAL I/O

1.1 Purpose

1.1.1 Model Development

The initial step in developing a design to support secure external I/O for a secure Multics is to develop abstract models for how external I/O is to be performed. From an engineering viewpoint, these models must provide adequate functionality to allow external I/O to be performed efficiently, economically, and conveniently. From a security viewpoint, these models must provide complete mediation of all references to information in the system virtual memory and to external I/O devices, must identify the functions performed by the protected kernel to ensure this mediation, and must lead to a kernel implementation that is simple enough to be certified by currently available methods.

This document presents <u>abstract models</u> that satisfy these requirements. These are not the only models possible. However, they do lead to designs and implementations that are minimally different from the current Multics implementation, and therefore have demonstrated their usefulness and feasibility in a real environment.

when?

Specifically, two different approaches to secure external I/O are modeled separately, because they provide fundamentally different primitive operations. For many types of external I/O, particularly those involving high-speed peripheral storage devices, high bandwidth channels and low-level program control need to be avalable at the user interface. For other types of external I/O, particularly communications-oriented I/O, simplicity of use and economic sharing of scarce system resources are the overriding requirements. These two sets of requirements are sufficiently dissimilar that a common model and mechanism to handle both would require excessive generality and complexity within the kernel.

1.1.2 Top-Level Kernal Design Description.

Given models that are sufficient, the next step in developing secure external I/O is to specify in some detail the semantics of kernel functions available at the kernel interface to uncertified active agents, such as Multics processes and device control code on auxiliary processors.

Descriptions of all necessary kernel functions are presented. The descriptions are intended primarily to demonstrate the sufficiency of the functions chosen, from both security and engineering viewpoints. Therefore, they include both interface details and some indications of expected use and implementation.

1.2 Scope.

1.2.1 External I/O only.

The models and descriptions apply only to external I/O, which includes the movement of information between the uncertified user environment (running in the virtual memory) and I/O devices outside the system security perimeter. Other I/O operations within the kernel, between kernel and user processes only, and between kernel and external I/O devices are not external I/O operations and are outside the scope of these models.

1.2.2 Networks not covered.

Secure communication and inter-computer networks are on the technical horizon. However, the functional and security requirements for integrating such networks into a secure Multics system are not yet sufficiently well understood to make modeling network functions productive. Therefore, there is no explicit mention of networks or network functionality.

Must conside
netwo

These designs do not, however, preclude the building of secure, multi-level or single level network functions on top of, or as an adjunct to the secure I/O designs presented.

2. REFERENCES

In preparation.

3. EXTERNAL I/O IN A SECURE MULTICS

3.1 General Definitions.

External I/O

All I/O requested by uncertified (non-kernal) software. For the purposes of this report, external I/O is split into two types: IOM I/O and SFEP I/O. A distinction may also be made between the two logical types of external I/O, namely, communication I/O and peripheral I/O.

IOM I/O

All I/O performed by the I/O Multiplexer (IOM). The IOM is peripheral-oriented (its protocols are optimized for peripheral control) but not restricted to peripherals.

SFEP I/O

All I/O performed by the secure front-end processor (SFEP). The SFEP is communication-oriented (its protocols are optimized for communication control) but not restricted to communication.

Communications I/O

All I/O performed for the purpose of communication between Multics and an intelligent (thinking, not smart) device or person. This includes terminals, networks, programmable terminals, etc.

Peripheral I/O

All I/O performed for the purpose of transfering stored information to (from) a recording medium from (to) Multics.

I/O Processor

A stored-program controlled machine specifically designed to control the transfer of date between main memory and I/O devices. It is a separate processor for reasons of efficiency; the main cpu can run in parallel with the I/O processor. In this report, the I/O processor includes all hardware between Multics and the actual device (i.e., it includes both the IOM and MPC's).

I/O Program

The sequence of instructions to be executed by an I/O processor.

I/O Process

The combination of an address space and execution point. The address space has a principle identifier, a security level, an integrity level, and a ring number. The objects in the address space are main memory locations and device locations. An I/O process is to the I/O processor as a Multics process is to the Multics processor.

I/O Device

Any electronic device capable of receiving and transmitting data to and from an I/O processor.

Secure I/O

I/O is secure if and only if any I/O program can be executed by the I/O processor without violating the security model (no unauthorized access to information, no unauthorized release of information).

Multiplexed I/O

A type of I/O in which several I/O devices share the same physical connection to the I/O processor, and the I/O processor can distinguish each of them. It is not multiplexed I/O if the I/O processor cannot distinguish each device.

3.2 General Security Principles.

3.2.1 External I/O devices inherently read/write from a security viewpoint.

How are the scarce devices handled?

We are specifically excluding from this study read-only or write-only devices because we do not believe they exist. In order to have effective control over a device (or to even communicate effectively with a device) some sort of ACK-NAK protocol is necessary. As soon as such a protocol is used, both reading and writing is taking place.

Explain

3.2.2 No simultaneous sharing of external I/O devices between processes.

This report restricts I/O devices to be attached by a single process at a time. This is done (1) because the current implementation does so, and (2) it does not seem to be necessary or useful. This restriction greatly simplifies the security model as well.

Yes, but is it reasonable

3.2.3 Validation of media handling (i.e., tape mounts) done outside system.

While the software may make some simple validity checks on tape reels and disk packs mounted by the operator, there is no foolproof way to be certain that the operator has not mounted the wrong one. Appropriate installation procedures will have to be used to enforce security constraints on tape reels and disk packs.

3.2.4 I/O program must be validated by hardware only.

The is the most stringent requirement of the design. Experience has shown that I/O programs are difficult to validate in software, and so even though we might, in theory, be able to prove that we can correctly validate all possible I/O programs, we have ruled it out. Another reason for requiring that the hardware do the validation is that it avoids duplication of function: the hardware is already performing (some) validation of the I/O program, and any software would have to duplicate (possibly incorrectly) these hardware checks. This requirement also forces us to develop a clear model of exactly what must be validated, and how, so that we can direct the hardware designers.

3.2.5 I/O design must be provable.

In order to meet Guardian's goal of a provably correct kernel, we must be able to prove the correctness of that part of the kernel that manages external I/O. The purpose of this report is to develop a design that meets this goal.

3.3 General Engineering Principles.

3.3.1 Two mechanisms needed for efficiency and compatibility: SFEP & IOM.

A fundamental point of this design is that two primitive mechanisms are needed to handle external I/O efficiently and with as little change as possible to the existing hardware and software of Multics. Any design that is less efficient that the present one is unacceptable in terms of performance, and any design that requires major hardware changes is unacceptable in terms of time and money. Multics currently uses ~~an I~~OM for high-speed, peripheral-oriented I/O, and a DataNet 355 for low-to-medium speed, communication-oriented I/O. A front-end processor is needed because the IOM cannot handle terminal channels, and because there is no other ~~GIOC~~-like device that can. But the front-end processor cannot handle peripheral I/O efficiently; it cannot handle the bandwidth required for disks and tapes.

This point is more substantial

undesirable but changes may be necessary for security

3.3.2 IOM provides efficient direct I/O.

The IOM primarily handles high-bandwidth, peripheral-oriented devices. It is capable of handling any device that can be plugged into it, whether it be communications or peripheral. Multics allows user-ring programs to write I/O programs that are executed by the IOM.

3.3.3 SFEP provides efficient communication-oriented I/O.

The SFEP handles low-to-medium bandwidth, communications-oriented devices. It is capable of handling any device that can be plugged into it, whether it be communications or peripheral. But it is definitely not designed for high-bandwidth devices. At this time it is an unresolved issue as to whether Multics will allow user-ring programs (on the Multics end) to write programs to be executed on the SFEP. But, no matter who writes the SFEP user-ring programs, the SFEP kernel will treat them all the same.

? How is this controlled?

Highly satisfying
IOM = high bw
SFEP = low bw

3.3.4 Few modifications to IOM.

The scope of the present Guardian effort does not allow the design of a completely new I/O processor. Since this is a prototype system, and since both time and the budget are severely constrained, we want to make only those changes that are absolutely necessary to the IOM hardware. The IOM is sufficiently flexible (and sufficiently correct) that only a few minor modifications will be needed. But they will be needed; without them we cannot achieve the goal of no software validation of I/O programs. It is hoped that these changes will be useful for the standard IOM, and will be incorporated into it, but this cannot be guaranteed.

3.3.5 No modifications to I/O devices.

While the I/O devices (in particular, the microprogrammed I/O devices) will have to be certified to be non-malicious, it would be uneconomical to propose a design that required modifying existing Honeywell peripherals and terminals. Fortunately, such changes are not necessary, and none are proposed.

3.3.6 Few changes to the user interface.

Since there is a large investment in existing software for Multics, and since these programs depend on the current user interface (the iox_ level, primarily) we want all changes to be invisible at this level. Thus, we will not introduce any incompatible changes to the user interface of I/O.

3.3.7 Other considerations.

We would like the design to be simple. We would like to remain compatible with the Standard Product Multics.

This section is
vague and repetitive.
Some supporting facts
would be helpful.
What are bandwidth capabilities
for IOM & SFEP?
What are bandwidth requirements
for communications (non-
communications external I/O
for Multics?

3.4 SFEP External I/O

This section describes basic concepts and presents a functional description of a kernel interface for performing external I/O between an uncertified Multics user process and an external I/O device via a front-end processor. The design supports user and supervisor interfaces that are highly similar to the current Multics user and supervisor interfaces for communications external I/O.

Seems to imply design choice will force some difference. What are they?

3.4.1 Scope

There is no fundamental reason why a secure external I/O mechanism using a front-end processor (FEP) could not be designed to handle both communications-oriented and peripheral-oriented external I/O. However, another secure mechanism not involving a FEP is available that is flexible and efficient (because of hardware validation) for peripheral external I/O. Therefore, the design presented in this section is intended only to support communications oriented external I/O operations.

The function of the security kernel in communications external I/O is that of trusted intermediary acting on behalf of and at the explicit request of two untrusted active agents: the uncertified user process in Multics and the uncertified device control code in the FEP. This functional description will focus on the Multics process-to-kernel interface only. The kernel-to-device control code interface is described elsewhere.

In the course of its other functions, the kernel performs, on its own initiative, many I/O-like operations with communications devices; namely dialup, login/dial, user-id authentication, security level validation, and telephone hangup processing. It is important to note that these activities are not external I/O because they do not involve the transfer of data originating outside the kernel between an uncertified user process and a device. Therefore, this design is not intended to support these activities. Clearly, these operations involve the transfer of data on which security decisions are based, and therefore must be done securely. However, these issues are substantially different from those of actual external I/O.

Why

3.4.2 Basic Concepts

This section is intended to provide an overview for the specific design described.

3.4.2.1 Kernel Provides Virtual Device Interface

The function of the kernel in FEP external I/O is to present a "virtual device" interface to a user process, as depicted in Figure 3.4.2.1.1. The user process may communicate with this virtual device via a restricted protocol of direct calls to kernel functions. The virtual device may communicate with the process by changing the process' execution state and/or control point in ways that are defined outside the external I/O path. The operations across this interface define the nature of Multics communications I/O via a FEP.

missing fig

?

This interface is somewhat asymmetrical. The process is viewed as being in control of the device. The kernel and/or device control code on the FEP must provide all buffering and priority routing necessary to support this interface.

3.4.2.2 All Device-to-Process Assignments Performed by Kernel

For communications external I/O there is no concept of a user process requesting to have a device assigned to it. This function is performed entirely within the kernel as part of login/dial processing.

What about
a network
connection?
Multics
Won't know
request connection
to network?
What about
RJE stations
remote printers?

3.4.2.3 Single-Level Communication

The communications external I/O interface supports only single-security-level communications. That is, a user process may always perform all available functions, particulary reading and writing, on a communications device assigned to it by the kernel. Put another way, the security and integrity levels of a communications device are always equal to the corresponding levels of the process using it.

3.4.2.4 No Sharing of Devices

A communications device is always in one of two states: not being used by any process, or being used by one process. A device may not be used by more than one process at a time. (Sharing of devices is accomplished outside the kernel via interprocess communication between Multics user processes.) The kernel guarantees that only one process may use each device at a time.

Networks

3.4.2.5 Multiple Devices Per Process

The kernel will allow one process to use more than one communications device simultaneously. All devices used by a process have the same security and integrity levels as the process.

3.4.2.6 Naming of Devices

The kernel performs all assignments of communications devices to processes. No process may use a device currently being used by another process. Thus, globally unique device names do not need to be visible to Multics user processes. Each process maintains its own list of (possibly) local names of communications devices it is using, to distinguish between the devices at the kernel interface. Efficiency ~~issue~~s dictate whether or not the device names at the kernel interface are in fact global.

Processes at different security levels can't use the same "global" name

3.4.2.7 Kernel Validates References

The kernel validates each reference to a device by a process by verifying that the process has been previously established as the using process of the device. The kernel validates each reference by a device to a process by directing all references by a device to its previously established using process.

3.4.2.8 Transparency of Functional Split

same para #

The split of function between the Multics and FEP kernels is invisible, except for performance, to the Multics user process and the FEP uncertified code.

The split of function between the kernel and FEP uncertified code is transparent to the Multics user process.

N.P. supervisor?

3.4.2.8 Multics - FEP Communication is Internal I/O

The management of channels and buffers for communication between the Multics and FEP kernels is hidden entirely within the kernel. Available hardware, and efficiency and code size and complexity issues determine the character of this interface.

Why? Double buff- may result in many cases?

3.4.2.9 Code Conversion Outside Kernel

All conversion betwen Multics standard ASCII character code and other character codes, canonicalization, escaping, and insertion of spacing and timing characters can be done outside the kernel. Efficiency issues determine how these functions are split between the Multics user process and the FEP uncertified code.

Good human engineering for login and authentication dialogues between the kernel and communications devices may require that some code conversion be performed within the kernel. The extent of common code and tables between this function and user code conversion depends on the size and certifiability of code conversion algorithms, and whether a layering is possible to allow some code conversion in the kernel.

3.4.2.10 Stream Orientation and Synchronization

The interfaces for reading and writing data are stream-oriented. That is, characters are read by the process in approximately the order input on the real device, and characters are output to the real device in approximately the order written by the process.

The data (read, write) interfaces are partly asynchronous in that read-ahead (input characters are buffered behind the kernel-to-process interface before the process requests them) and write-behind (output characters are buffered until they can actually be written) are supported.

The control interfaces are highly synchronous. There is no notion of queued control operations - they take effect before the requesting process regains control.

3.4.2.11 Read Delimiters

For data reading operations, the kernel will recognize a "delimeter" character to delimit logically separate units of input so that they may be read one at a time by the Multics process.

How will
devices that
are not necessa
line oriented fi

Figure 3.4.2.1.1 The kernel presents a virtual device interface to a user process

3.4.3 Functional Description

3.4.3.1 Attributes Maintained by Kernel

The kernel maintains several security-related attributes used ~~in~~ the validation of communications external I/O operations.

which SFEP or Multics

process_id - This uniquely identifies a process at any instant of time, and is constant for the life of the process.

device_id - This uniquely identifies a device (or its connection point to the system for dial-up lines), and is constant for the duration of its use by a process (although it may in fact be constant for a longer time). (This ~~may not be~~ the process-local name for the device.)

Unclear! Is the id a system-wide logical name or a process local name or?

using process_id for each device_id - This is the routing information against which the kernel validates I/O operations.

event_channel_number for each device_id - This is the IPC event channel over which the device may stimulate the using process.

Process ids and device ids may be visible outside the kernel. There is no reason why the using process id or event channel number of a given device id should be visible outside the kernel.

Other security and access control related attributes of processes and devices are maintained by the kernel, but are not used in the validation or routing of actual I/O operations.

3.4.3.2 Initial State

Process and device may not communicate with each other until the kernel has completed the setup of an initial state. This state holds until the process or device makes a non-I/O request to the kernel to end the assignment.

The initial state is defined by the following:

1. The using process id is established for the device id.
2. The event channel number is established for the device id.

3. The process has a name by which it may refer to the device at the kernel interface.

4. The process has a handler for signals (quits).

5. For code conversion and mode functions performed by the kernel, initial tables and values have been established.

? are these kernel related

6. A read delimeter has been chosen for this device.

3.4.3.3 User Process Operations on Virtual Device

The Multics user process may perform the following functions on the virtual device:

read (some data)

write (some data)

abort (some concurrent data read or write)

unassign (terminate the connection)

control

status

These are described in detail below both informally and in a Parnas-like verbal notation.

All functions take a device name as a parameter, and have a common exception condition for validatior abbreviated as NOT-ASSIGNED, defined as either the device name supplied is not valid (does not correspond to a device), or the calling process is not recorded in the kernel as the using process for this device.

3.4.3.3.1 Read

This OV-function reads some data from the virtual device. The maximum amount of data to be read is specified as an argument. The function returns to the caller any pending input from the device up to and including the first read delimeter character encountered. If the kernel does not yet have a complete unit of data yet (no read delimeter in its buffer for this device yet), the call returns with no data. If the supplied buffer is smaller than the first unit of data, as much data as will fit is read. (It is assumed that the supervisor will map this interface into the more natural user interface read call that returns only when a unit of input data is ready.)

More formally

OV-function: read

parameters: device name
buffer address
buffer size

security check?!

exceptions: NOT-ASSIGNED
no input from this device yet
zero length buffer

values: all characters of pending input up to and including the first delimeter, or buffer size, whichever is smaller.

Is this a value or an effect? If it's a value why are buffer addr & size parameters

the number of characters read

a status code indicating whether the buffer was large enough.

effect: the number of characters read are discarded from kernel buffers.

is this effect visible at the kernel interface.

3.4.3.3.2 Write

This O-function writes a buffer of data to the virtual device. This function returns to the caller when the virtual device (i.e., kernel buffers) has the data.

security check?!

O-function: write

parameters: device name
buffer address
buffer size

exception: NOT-ASSIGNED

effect: the buffer of characters is queued behind characters for previous write calls, and is eventually output to the device.

seems to say there is no observable effect

3.4.3.3.3 Abort

This generic O-function is really three similar O-functions to abort pending read operations, write operations, or both. This function recognizes that actual input and output operations go on in parallel with the intended or requesting process, and aborts them, flushing out any queued data. Whether these functions extend to buffering outside the kernel in the FEP depends on the nature of the FEP kernel interface.

O-functions: abort_read
abort_write
abort_all

parameter: device name

exception: NOT-ASSIGNED

effect: all pending operations of the indicated type for this device are stopped, all related queues are flushed, and kernel resources used by these operations are freed.

Is a non-kernel abort routine [illegible]

3.4.3.3.4 Unassign

This O-function is a request by the process to the kernel to destroy the communication path with a device. It is included here because it is an explicit request made by the process (unlike login/dial etc. that are requested by a device before it is connected to the user process environment).

Since a device may be assigned to only one process at a time, this function returns the device to the state where it must re-negotiate with the kernel to be assigned to a new process. Two versions of this may be necessary, a strong one which also hangs up the telephone line and/or powers down the device, and a weak one that simply returns the device to the kernel without physically disconnecting it.

Formally,

O-function: unassign

parameter: device name

exception: NOT-ASSIGNED

effect: the device no longer has a using process or user process event channel associated with it.

The device may be hung up or powered off.

3.4.3.3.5 Control

This generic O-function incorporates all mode, translation table, and device control functions supported by the kernel. (Other device control functions can be coded in data interpreted only by non-kernel code at either end of the kernel.)

Be more specific

O-function: control

parameters: device name
control operations and data

exceptions: NOT-ASSIGNED
invalid or unsupported control operation

effect: the indicated control operation(s) is performed

3.4.3.3.6 Status

This generic V-function may actually be several V-functions to return parts of status information about a particular device assigned to the process, such as current modes, translations, carriage and paper positions, write-behind and read-ahead status, etc.

specifically, what

Are these necessary for kernel? Will require kernel to know about all terminals (e.g. what characters ~~print~~ do/don't print).

V-function: status

parameter: device type

exception: NOT-ASSIGNED

values: status information for the particular request.

Security check

3.4.3.4 Virtual Device Operations

Even though a communications device is largely under control of the user process, it still must be able to stimulate the process at its own initiation in order to indicate situations that the process must respond to.

This stimulation is highly restricted, and is limited to

- informing the process of a pending unit of input, which the process should read when it gets a chance; and
- indicating exceptional events which the process should be made aware of instantaneously (in virtual time).

The common exception for these is NO-PROC, defined as the kernel does not have a valid using process id recorded for this device, either because none has been assigned, or because the process is dead. Generally, this exception will cause the kernel to become involved in a non-I/O capacity.

Both these operations pass control to pre-arranged non-kernel code in the using process to perform the operations necessary to sort out the reason for the stimulus.

For abstraction purposes, these are best viewed as O-functions performed by the virtual device on the process.

The kernel "validates" both these O-functions by directing them always to the using process for the device. (There is no way for a device to indicate any other process.)

3.4.3.4.1 Wakeup

This O-function queues an IPC wakeup for the using process over the recorded event channel in response to the receipt of a read-delimeter from the device. The standard response (presumably in non-user supervisor code) is for the process to issue one or more read OV-functions on the virtual device associated with the event channel on which the wakeup was received.

Formally,

O-function: wakeup (no parameters)

exception: NO-PROC

effect: queue an IPC wakeup for the using process over the pre-specified event channel.

3.4.3.4.2 Signal (Quit)

This O-function causes the process to immediately (in virtual time after all critical sections in the supervisor are completed) execute a well-defined block of code to handle this signal. The actual block of code invoked may change with changing process states and desired interpretation of signals, but one such block is always defined.

The standard response for a process that may be the using process for several devices is to first issue status V-functions to determine which device sent the signal, and then to perform pre-defined actions associated with the process state. (A process that knows it is using only one device can skip the device identification step.)

vague

Formally,

O-function signal (no parameters)

exception: NO-PROC

effect: cause the process to execute its signal handler immediately (in virtual time).

3.5 IOM External I/O.

This section describes, in turn, some definitions that are unique to the description of the IOM, some engineering considerations that are unique to the IOM, an abstract model of the operation of the IOM, an implementation of the model, performance estimates for the implementation, and an evaluation of the impact on existing programs (both within the supervisor and outside it).

~~3.5.1 IOM Definitions.~~

~~See section 3.1, General Definitions.~~

~~3.5.2 IOM Principles.~~

~~See sections 3.2 and 3.3, General Security Principles, and General Engineering Principles.~~

3.5.3 IOM Model.

This section of the report describes an abstract model of secure external I/O. Initially, a very simple model is described; an I/O processor that serves a single device, and executes a single I/O program at a time. The concept of a reference monitor is introduced, to validate all references to main memory by the I/O processor. We show that, no matter what the I/O program does, it cannot reference any portion of main memory outside the limits enforced by the reference monitor.

Not really?

Next, the model is extended to cover an I/O processor that can serve many (non-multiplexed) devices securely. This model is in direct correspondence to the operation of the IOM. We show what values must be associated with each device channel, and what tasks must be performed when the I/O processor switches from channel to channel.

Finally, the model is extended to cover multiplexed I/O. We add the concept of a device number reference monitor, and show how this enforces access to a single device on a multiplexed channel.

3.5.3.1 Description of IOM Model.

The elements of the model are a Multics process, an I/O buffer segment in Multics, an I/O program in the buffer segment, the Multics kernel, the IOM reference monitor, the IOM itself, and the device.

The Multics kernel maintains a table that describes each device (listing all of its attributes, and its temporary qualities). [THESE SHOULD BE DESCRIBED IN DETAIL]. Every device has an associated I/O buffer segment (located in the user's process directory). The user constructs the I/O program in the buffer segment.

How?

The only main memory addressable by the I/O program is the buffer segment; it must read (or write) directly into the buffer segment itself. It is the program's own responsiblilty not to overwrite itself.

The user calls the kernel, passing the device to be started, and the offset of the I/O program to be used.

The kernel validates that the device is indeed attached to this process, and that the device is not currently running (another I/O program).

?

The kernel then loads the reference monitor with the offset and length of the (wired) buffer segment in main memory, and the device number.

The kernel then starts the I/O program.

As interrupts are received from the device, the kernel sends wakeups to the user's process. Status from the device is stored directly into the buffer segment.

3.5.3.2 Top-Level Specification of IOM Model.

This section describes, in (informal) Parnas-type specifications, the functions available to a process executing on Multics (first section), and a process executing on the I/O processor (second section).

3.5.3.2.1 Functions available to a Multics process.

Assign (devno, uproc)

Description:
Assign a device to a process.

```
Exceptions:
        no_access! check_security (uproc, devno)
        no_access! check_integrity (uproc, devno)
        no_access! check_acl (uproc, devno)
        already_assigned! device (devno).assigned = True
Effect:
        device (devno).uproc := uproc
        device (devno).assigned := True
        device (devno).attached := False
        device (devno).buffer_seg := null
        device (devno).buffer_size := 0
        device (devno).buffer_absaddr := 0
        device (devno).event_chn := 0
        device (devno).status_offset := 0
        device (devno).running := False
```

Attach (devno, buffer_size, event_chn, status_offset)

What is event_chn?

```
Description:
        Attach a device to a process
Exceptions:
        not_assigned! device (devno).uproc ^= cur_proc
        already_attached! device (devno).attached = True
        invalid buffer size for this device
Effect:
        device (devno).event_chn := event_chn
        device (devno).buffer_seg := create_seg (buffer_size)
        device (devno).buffer_size := buffer_size
        device (devno).status_offset := status_offset
        device (devno).attached := True
```

Is buffer segment number returned?

Detach (devno)

```
Description:
        Detach a device from a process
Exceptions:
        not_attached! device (devno).uproc ^= cur_proc
        not_attached! device (devno).attached = False
        device_running! device (devno).running = True
Effect:
        destroy_seg (device (devno).buffer_seg)
        device (devno).buffer_seg := null
        device (devno).attached := False
```

Unassign (devno)

Description:
Unassign a device from a process.
Exceptions:
not_assigned! device (devno).uproc ~= cur_proc
not_assigned! device (devno).assigned = False
not_detached! device (devno).attached = True
Effect:
device (devno).uproc := 0
device (devno).assigned := False

Connect (devno, io_program_offset)

Description:
Start I/O program on a device.
Exceptions:
not_attached! device (devno).uproc ~= cur_proc
not_attached! device (devno).attached = False
device_running! device (devno).running = True
Effect:
device (devno).running := True
wire_io_segment (device (devno).buffer_seg)
device (devno).buffer_absaddr := absaddr
(device (devno).buffer_seg)
mailbox.base := device
(devno).buffer_abs_start
mailbox.bounds := device (devno).buffer_length
mailbox.status_offset := device
(devno).status_offset
mailbox.devno := devno
mailbox.program_offset := io_program_offset
start_device (mailbox)

How does user specify the virtual address from/ to which data is to be transferred. Buffer seg is never a parameter or a return value.

3.5.3.2.2 Functions available to a process on the I/O processor.

Transfer (mailbox, target_address)

Explain somewhere what mailbox is.

Description:
Change program counter of I/O program.
Exceptions:
address_negative! target_address < 0
address_too_big! target_address + mailbox.base
> mailbox.bounds
Effect:
cur_io_pc := target_address

Generic_Device_Operation (mailbox, devno, operation)

Description:
A typical I/O instruction that affects the device.
Exception:
wrong_device: mailbox.devno ^= devno
Effect:
perform operation

Transfer_to_Device (mailbox, devno, address, tally)

Description:
Typical memory-to-device transfer.
Exception:
wrong_device: mailbox.devno ^= devno
negative_tally: tally < 0
Effect:

```
for offset := tally to 0 by -1 begin
     temp_address := address + offset
     value := Fetch (mailbox, temp_address)
     ship_to_device (devno, value)
end
```

Doesn't this specify a backward read from memory?

Transfer_to_Memory (mailbox, devno, address, tally)

Description:
Typical device-to-memory transfer.
Exception:
wrong_device: mailbox.devno ^= devno
negative_tally: tally < 0
Effect:

```
for offset := tally to 0 by -1 begin
     temp_address := address + offset
     value := get_from_device (devno)
     Store (temp_address, value)
end
```

Backward read from device?

Terminate_Program (mailbox)

Description:
Stop I/O program on a device.
Exceptions:
none.
Effect:

```
devno := mailbox.devno
msg := "term" || devno
Store_Status (mailbox, msg)
Interrupt (mailbox)
unwire_seg (device (devno).buffer_seg)
device (devno).buffer_absaddr := 0
```

```
            device (devno).running := False
```

Store_Status (mailbox, status)

```
    Description:
            Store status from device in buffer segment.
    Exceptions:
            none.
    Effect:
            status_address := mailbox.base + mailbox.status_offset
            Store (mailbox, status_address, status)
```

Interrupt (mailbox)

```
    Description:
            Map interrupt into wakeup
    Exceptions:
            none.
    Effect:
            devno := mailbox.devno
            pid := device (devno).uproc
            chn := device (devno).event_chn
            msg := 0
            send_wakeup (pid, chn, msg)
```

Where is this function described? The wakeup on pg 23 has no parameters

Store (mailbox, virtual_address, value)

```
    Description:
            Store into main memory.
    Exceptions:
            negative_address! virtual_address < 0
            address_too_big! virtual_address + mailbox.base > mailbox.bounds
    Effect:
            absolute_address := virtual_address + mailbox.base
            write (absolute_address, value)
```

Fetch (mailbox, virtual_address) Returns (value)

```
    Description:
            Read main memory.
    Exception:
            negative_address! virtual_address < 0
            address_too_big! virtual_address + mailbox.base > mailbox.bounds
    Effect:
```

```
absolute_address := virtual_address + mailbox.base
value := read (absolute_address)
```

3.5.3.3 Block Diagram of IOM Model.

FUNCTIONAL BLOCK DIAGRAM OF IOM MODEL

3.5.4 IOM Implementation.

In preparation.

3.6 Performance Evaluation Estimate.

In preparation.

3.7 Impact on Existing I/O Programs.

In preparation.