ARPA ORDER NO.: 189-1

R-562-ARPA

August 1971

The ISPL Machine: Principles of Operation

R. M. Balzer

A Report prepared for

ADVANCED RESEARCH PROJECTS AGENCY

Rand
SANTA MONICA, CA 90406

35

This research is supported by the Advanced Research Projects Agency under Contract No. DAHC15 67 C 0141. Views or conclusions contained in this study should not be interpreted as representing the official opinion or policy of Rand or of ARPA.

DOCUMENT CONTROL DATA

1. ORIGINATING ACTIVITY The Rand Corporation	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP
3. REPORT TITLE THE ISPL MACHINE: PRINCIPLES OF OPERATION	
4. AUTHOR(S) (Last name, first name, initial) Balzer, R. M.	

5. REPORT DATE August 1971	6a. TOTAL NO. OF PAGES 36	6b. NO. OF REFS. 2
7. CONTRACT OR GRANT NO. DAHC15 67 C 0141	8. ORIGINATOR'S REPORT NO. R-562-ARPA	
9a. AVAILABILITY/LIMITATION NOTICES DDC-A	9b. SPONSORING AGENCY Advanced Research Projects Agency	

10. ABSTRACT

The first of a series conceptually describing the Incremental System Programming Language computing system, an integrated environment for multiuser research programming. The ISPL language and machine are jointly designed, with hardware providing the control and scheduling facilities traditionally handled by Job Control Language and other software. Close correspondence between program statements and machine actions makes for clarity and efficiency and facilitates incremental compilation, which in turn allows on-line, interactive programming and debugging. During postfix program translation, ISPL inserts NEW STATEMENT operators that define interruptible points. User address spaces are carefully segregated. Separately accessed memory areas are assigned in logical units, with pointers. Most programs and data remain in virtual memory; only those portions of program and data actually referenced are contained in real memory. Resources are allocated by machine primitives called semaphores, which may also carry data. Data semaphores compose Ports, which provide hardware/software/user communications (described in R-605.)

11. KEY WORDS

Computers
File Structure and Management
Computer Programming
ISPL

ARPA ORDER NO.: 189-1

R-562-ARPA

August 1971

The ISPL Machine: Principles of Operation

R. M. Balzer

A Report prepared for

ADVANCED RESEARCH PROJECTS AGENCY

Rand
SANTA MONICA, CA. 90406

PREFACE

This report describes the Incremental System Programming Language (ISPL) machine, which was designed specifically to run the ISPL language. Together, the language and the machine comprise a complete system for producing a programming laboratory at Rand. The ISPL machine directly executes the postfix representation of the ISPL language and includes many functions, such as page-table maintenance, normally found in software. For a clear picture of the ISPL system, this report should be read in conjunction with its companion paper, R-563-ARPA, *The ISPL Language Specifications*. However, both reports should be viewed as specification documents only, as the system has not yet been implemented.

Work on ISPL was sponsored by the Department of Defense's Advanced Research Projects Agency (ARPA) as an integral part of both Rand's and the client's overall program to explore current computer technology. The present report should be of interest to those concerned with machine design and the integration of hardware and software systems.

SUMMARY

The design of the ISPL machine has been integrated with the design of the ISPL language Together, they comprise a complete system for producing a programming laboratory. The facilities incorporated in the machine allow for the direct execution of the ISPL language at the postfix level. This simplifies compilation of the language and facilitates incremental compilation. The postfix execution unit includes debugging capabilities for dynamic type checking and pointer verification.

The addressing structure is based on variable-length segments accessed through pointers, which also contain read/write capability information. Remote segments enable the sharing of segments and of data structures, which themselves contain pointers.

The machine has operations for scheduling core and central processing unit resources, and handles all interrupts and communications through semaphores, which exist as machine primitives.

ACKNOWLEDGMENTS

Many of the ideas contained herein arose during the ISPL study group meetings. As such, it is impossible to individually credit each idea, but my thanks to the members: Richard Bisbey, Rod Fredrickson, Bill Josephs, Larry Lewis, and Tom Wall.

My special thanks to Bill, who started the project with me and helped crystallize many of the notions upon which the ISPL machine and language are based.

CONTENTS

Published by The Rand Corporation

I. INTRODUCTION

The Incremental System Programming Language (ISPL) machine has been expressly designed to execute programs written in ISPL. Our goal was to execute these programs efficiently, with as much dynamic error checking as possible and so that they resembled as closely as possible the ISPL source language. This goal is designed to foster an environment in which *all* our programming work can be done in ISPL because (1) it is efficient, (2) it has powerful debugging techniques built in; and (3) its close correspondence to machine actions makes errors easily interpretable in source form and provides easy and efficient handling of incremental compilation of programs to allow on-line, interactive programming and debugging.

The third major component, after the language and the machine, is the control program to coordinate the two. The language design specifically includes the facilities needed by the control program, and the machine is designed to easily express these control functions. In addition, many facilities in both the language and the control program that have traditionally been implemented in software have been designed as part of the ISPL machine.

This intense integration simplifies the design, implementation, efficiency, and understanding of the system. However, it makes the description of one of these components without the others virtually impossible. Thus, *The ISPL Language Specifications* [1] should be considered as a companion report. Each relies heavily on the other.

II. PROGRAM EXECUTION

The control program schedules units (called PROCESSES) for execution on the ISPL machine. A PROCESS is a set of cooperating, independent control paths (called TASKS) through programs sharing a common addressing space for programs and data. The ISPL machine is responsible for scheduling and controlling the TASKS within a PROCESS; this is described in Sec. V. Processes are created via a form of the INITIATE_PROCESS machine operation, which causes a series of control blocks to be formed that contain all the necessary context and control information for the process and each of its tasks.[†]

To start a process in execution, the DISPATCH_PROCESS operation is used, which specifies a PCB to be dispatched. The machine context of the current process (the one that issued the DISPATCH_PROCESS operation) is stored in its PCB, the dispatched process is marked in the DISPATCHED field and its machine context is loaded into the ISPL machine. This machine context consists of:

1. Process number, which indicates which process is running.
2. Segment table pointer, used for translating virtual addresses (see pp. 7-11).
3. Top of stack pointer, which indicates the location of the top of the stack.
4. Interpretation pointer, which indicates the procedure context in which operands are to be interpreted (see p. 4).
5. Syllable pointer, which indicates where execution is to be resumed.

After completion of the DISPATCH operation, the dispatched process is running and syllables are fetched and executed sequentially, starting at the location specified by the syllable pointer. These syllables are the basic instructions of the ISPL machine. They are either operands or operators and are the Polish Postfix translation of the corresponding source statement. A syllable has the following format:

[†]The exact format of this information is described in Sec. V; until then, we refer to these blocks collectively as a Process Control Block (PCB).

Operator syllable (1 or 2 bytes)

operator bit	1 bit
operator type	7 bits

If operator type = S'7F', then the next byte contains the operator type (all 8 bits are concatenated with a high-order bit (value, 1) to form the operator type).

Operand syllable (2 to 5 bytes)

operand bit		1 bit
value or address desired		1 bit
location of operand		2 bits
literal	00	
offset in static	01	
offset in current record	10	
offset in record	11	
data type of operand		4 bits

The second part of the operand syllable is determined by the LOCATION OF OPERAND value in the first part. It will be one of the following:

literal	
literal value	8 bits
offset in static	
offset value	16 bits
offset in current record and offset in record	
record type of record in which operand occurs	8 bits
offset within record	
if high-order bit is off	8 bits
if high-order bit is on and next highest order bit is off	16 bits
if high-order bit is on and next highest order bit is on	24 bits

The action of the ISPL machine when executing a syllable depends upon the type of syllable and the options selected within it. In general, operand syllables cause either the address or value (as determined by the address-or-value bit within the operand) to be added to the top of the process stack. Operator syllables also cause the corresponding ISPL-machine operator to be executed, which uses and removes the top N (usually two) operands from the process stack and replaces them with M results (usually one or zero).

Execution of the process continues until (1) it completes, (2) an interrupt occurs, or (3) it waits for an event that has not yet occurred. In each case, the machine context of the running process is saved. The first two cases are handled by the interrupt-processing mechanism in the ISPL machine (described in Sec. V). In the third case, the process that dispatched the running process is dispatched at the syllable following the DISPATCH_PROCESS operator, which it used to dispatch the waiting process.

Consider a process that has been dispatched and is now running. We describe the actions of the ISPL machine in decoding and accessing each of the operand syllables.

First, we assume the VALUE_OR_ADDRESS bit specifies address.

LITERAL

A program error occurs. Literals must be specified as values.

OFFSET IN STATIC

The 16-bit offset is pointer-added (pointer addition is defined on p. 12) with the STATIC SEGMENT pointer from the PCB to form the required address and is pushed onto the top of the PROCESS STACK.

OFFSET IN CURRENT RECORD

The INTERPRETATION POINTER in the PCB points at a procedure entry in the PROCESS STACK; this entry contains the address of the DISPLAY for that procedure. The DISPLAY contains the address of the current member of each record declared in the compilation in which the procedure occurs. When a procedure is entered, its DISPLAY is initialized to that of its calling program if they are in the same compilation. Otherwise, its DISPLAY is set to all NULL. When a process is dispatched, the INTERPRETATION POINTER is used to obtain the address of the associated DISPLAY. This address is kept in the CURRENT DISPLAY REGISTER.

The 8-bit record_type is multiplied by 4 and pointer-added to the contents of the CURRENT DISPLAY REGISTER; the contents of the cell referenced by the resulting pointer is accessed and pushed onto the top of the PROCESS STACK. This is the reference to the current instance of the specified record_type.

Processing now continues as for OFFSET IN RECORD.

OFFSET IN RECORD

The 8-bit record_type in the operand SYLLABLE is compared with the 8-bit record_type in the pointer at the top of the PROCESS STACK. If they do not agree or if a pointer is not at the top of the PROCESS STACK, a program error occurs.

If the record_types agree and a pointer is at the top of the stack, the offset specified in the operand SYLLABLE is pointer-added to the value at the top of the stack and replaces it at the top of the stack.

* * * * *

In each of the cases above, except for the illegal case of literals as addresses, the resulting value at the top of the stack is a pointer and is so marked by the 4 control bits in the stack entry. The record_type of the pointer value is set to the primitive datatype specified by the 4-bit data_type specified in the first part of the operand SYLLABLE.

If the VALUE_OR_ADDRESS bit specifies that the value is desired, the processing for each of the cases is as follows.

LITERAL

The 8-bit literal (as the low-order 8 bits of the 32-bit stack entry) is pushed onto the top of the stack. The control bits of this entry are the data_type bits specified in the operand SYLLABLE.

<u>OFFSET IN STATIC</u>

<u>OFFSET IN CURRENT RECORD</u>

<u>OFFSET IN RECORD</u>

In each case, after processing the individual cases as described above for address operands, the resulting pointer at the top of the stack is replaced by the value referenced by this pointer. The control bits of this entry are the right-most 4 bits of the pointer's record_ type (the other 4 bits should all be zero).

III. ADDRESSING

Section II completes the description of the logical addressing scheme for the ISPL machine. This would suffice if a REAL, as opposed to a VIRTUAL, addressing scheme were used. The ISPL machine has been designed as a multiuser research machine in which programs cannot be assumed to work correctly. Therefore, the address spaces of separate users must be kept disjoint. Furthermore, a mechanism is needed to more effectively use one of the machine's scarcest resources--real memory. A number of VIRTUAL addressing schemes satisfy the first requirement. Those schemes that also divide this virtual address space into separately accessed pieces allow the machine to run a program with only the actually referenced portions of a program and its data in real core. This is especially important to us because our programs tend to be large compared with our available real core. Within these major constraints, we based our choice of a virtual addressing mechanism upon the following critical requirements:

1. Units of physical addressing space should correspond to logical units of a user's program and/or data.
2. Addresses are a separate data type and have their own operators; they are not a form of integer data.
3. The address spaces of separate processes must be protected from each other while still allowing data and program sharing.
4. The addressing structure must allow subsystems to be built that have direct control of their user's address space and allocation, but that cannot affect other users' address space.

All addresses, whether in the stack or in pointer variables, are virtual addresses and must be transformed into real addresses before being used. Each virtual address is part of a pointer value, which has the following format:

8 bits	2 bits	22 bits
record type	read/write capability	virtual address

For the moment, we assume that the virtual-address portion is divided into two parts--a segment number and an offset (in bytes) within that segment.

To transform the virtual address, the ISPL machine uses the segment number to index a table, the SEGMENT TABLE, pointed to by the SEGMENT_TABLE_POINTER in the process' PCB.[†] Each entry in this table is a double word, as shown in Fig. 1.

In the first case, REAL SEGMENT, the ISPL machine multiplies the SEGMENT SIZE field of the entry by 8 to convert it to a byte length and compares this with the offset in the virtual address. If it is larger, a segment-overflow program-error occurs. If not, the REAL SEGMENT BASE field of the entry is multiplied by 8 to convert it to a byte address and the offset in the virtual address is added to this value to create the resultant real address.

If a store operation is being done and the SEGMENT TYPE is DATA AND PROGRAM UNMODIFIED, the SEGMENT TYPE is changed to DATA AND PROGRAM MODIFIED.

The second case is the PSEUDO-SEGMENT. The transformation is the same as in the first case except that the resultant address is a virtual address in the same address space and must be decoded again as explained in this section. PSEUDO-SEGMENTS are segments overlayed on all or part of a REAL or REMOTE segment; they are created by a user for his own convenience, usually protection. They never represent separate addressing space and play no part in core allocation, scheduling, or swapping.

In the third case, the segment-table entry is a REMOTE SEGMENT. This indicates that the resultant address of the transformation will not be a real address but a virtual address in the address space of the process specified by the REMOTE PROCESS NUMBER field in the entry. The read/write capability specified in the entry is combined, by the rules of read/write capability combination described above, with the

[†]See p. 11 for a discussion of how this address is transformed into a real address.

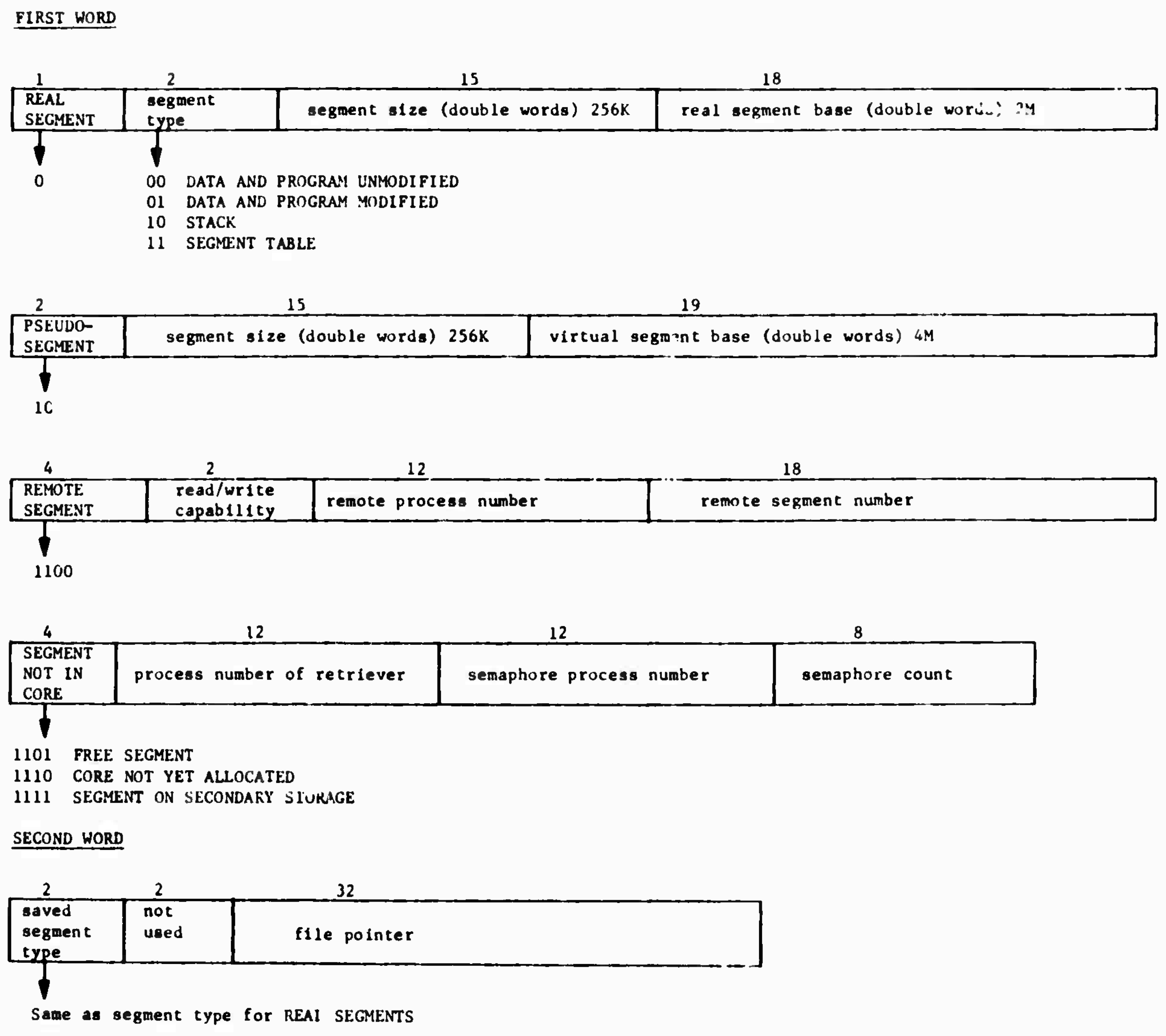
FIRST WORD
1
2
15
18
REAL SEGMENT
segment type
segment size (double words) 256K
real segment base (double words) 2M
0
00 DATA AND PROGRAM UNMODIFIED
01 DATA AND PROGRAM MODIFIED
10 STACK
11 SEGMENT TABLE
2
15
19
PSEUDO-SEGMENT
segment size (double words) 256K
virtual segment base (double words) 4M
10
4
2
12
18
REMOTE SEGMENT
read/write capability
remote process number
remote segment number
1100
4
12
12
8
SEGMENT NOT IN CORE
process number of retriever
semaphore process number
semaphore count
1101 FREE SEGMENT
1110 CORE NOT YET ALLOCATED
1111 SEGMENT ON SECONDARY STORAGE
SECOND WORD
2
2
32
saved segment type
not used
file pointer
Same as segment type for REAL SEGMENTS

Fig. 1--Format of Segment Table

read/write capability of the pointer value being transformed, to form the resultant read/write capability. The REMOTE PROCESS specified becomes the base for further transformation; the offset in the pointer value is then combined with the remote-segment number to form the resultant virtual address; and the transformation, using the REMOTE PROCESS's segment table, continues as above.

The remote-segment concept is very important to the ISPL machine's addressing mechanism. It allows two arbitrary processes to share programs and data, including pointer-linked data, without special programming conventions or restrictions. Remote segments are built by the ISPL machine whenever a pointer is passed (i.e., stored) from one addressing space to another. When such a transfer occurs between two processes that use different segment tables, the ISPL machine creates a remote segment in the receiving segment table and changes the passed pointer to reference the newly created remote-segment entry.

In order to tell when pointer values are stored in a different addressing space, an indication of its addressing space must be kept with each pointer value. For normal pointer values--those for which the initial address transformation is not remote--the indication is implicit in the control bits that indicate the entry is a pointer value. For remote pointer values--those for which the initial address transformation is remote--the control bits indicate a new stack type (remote pointer value) and the next entry indicates the process number of the process from whose addressing space the pointer value was obtained. Before a pointer store is done, a remote segment is created as described above if the process number of the pointer value and the process number of the address space in which the destination is real (i.e., not remote) are different.

Pointer comparison is always done by comparing real addresses; therefore, it is insensitive to remote segments. The only other pointer operation is POINTER FOLLOWING (picking up successive pointers through consecutive OFFSET IN RECORD operand SYLLABLES); it is also not modified. However, because all address transformations must begin with the process' addressing space from which the pointer value was obtained, the decoding mechanism must handle both normal and remote pointers.

Because both segment tables and stacks utilize all 36 bits of a word, the ISPL machine contains special operations to allow manipulation of these segments under program control. These operations are detailed on pp. 14-16; the relevant issue here is that only the operation that frees the entry involves remote segments. Thus, processes can neither create nor modify these crucial links between address spaces.

Finally, in the last case, the segment-table entry is a SEGMENT NOT IN CORE entry. If the entry is a FREE SEGMENT, a program error, ADDRESSING NON-EXISTENT SEGMENT, is generated. Otherwise, the segment is either on secondary storage or has not yet been allocated. In either case, real core must be allocated to the segment before the process can continue. Therefore, the ISPL machine issues a P operation for the dispatched process on the semaphore in the segment-table entry and generates a SEGMENT REQUESTED interrupt for the process that RETRIEVED the real core allocated to the segment (see pp. 15-16).

The SEGMENT TABLE POINTER in the PCB represents the real-core address of the segment table. Since all address transformation is through the segment table, a method of addressing it cannot require information within it. The simplest such method is the use of an absolute address. Since entries in the process' segment table represent all real core GIVEN to a process, it does not make sense to RETRIEVE the segment table if it contains any real core because that real core would not be retrievable or usable without the segment table. Thus, two special segment-table operations are defined to control the allocation of segment tables, MAKE PROCESS INACTIVE and MAKE PROCESS ACTIVE; these are explained on p. 17.

One final complication remains. With only 22 bits available for segment numbers and offsets, no fixed division allows enough flexibility for a wide range of applications. We adopted the following mechanism as a more flexible alternative. The 22-bit field is dynamically divided into the segment number and the offset portions on the basis of actual need, as evidenced by the largest segment number actually used. Eventually, as larger segment numbers are used, the binary representation of the largest will not fit into the allocated portion of the 22-bit field. Therefore, the ISPL machine dynamically increases the portion allocated to segment numbers. In order to preserve the validity of pointers already generated with the old boundary, the bit representation of the segments and offsets has the low-order end of each number at the outside of the 22-bit field and the high-order end toward the center. Thus, as the boundary dynamically shifts, it has no effect on existing pointer values because high-order zeros are either being added or deleted from the two parts. This remains true as long as the total number of bits needed to represent the maximum segment number and offset used is not greater than the 22 bits available.

If this condition is exceeded, an ADDRESS SPACE OVERFLOW program error occurs. The ADDRESS BOUNDARY REGISTER for each process is kept in its PCB and used to decode all pointer values occurring in its address space. The maximum-length segment used is kept in the header of the segment table. Since the maximum-size segment is 256K bytes, the ADDRESS BOUNDARY REGISTER is initialized to provide 18 bits for the offset and 4 bits for the segment number.

Finally, when an offset is added to a pointer, i.e., POINTER ADDED, a SEGMENT OVERFLOW program error occurs if an overflow occurs from the offset portion.

The read/write capability is a discrete value from the read_write_capability range consisting of the values read_only, read, read_write, and modal. These values are given in decreasing order of restrictiveness; therefore, in following a pointer chain or path, the resulting read/write capability is the more restrictive of the accessing and the accessed capability, as for example, in the pointer chain

```
P1(MODAL)->P2(READ_WRITE)->P3(READ_ONLY)->
        P4(READ_WRITE)->J
```

In this example, the values in parentheses indicate the read/write capability of the pointers. Some implicit pointer is used to access P1, and we assume its capability is modal. P2 is also accessed with modal capability, P3 with the more restrictive read_write capability, P4 with the most restrictive read_only, and J with this same read_only capability even though P4 has read_write capability. Thus, protection can be assured by starting with, or encountering, the proper capabilities in a pointer chain or path.

There is one exception to these precedence relationships: when a read capability encounters a read_write capability, it becomes a read_write capability. Thus, local read(only) protection can be given that, via an appropriate pointer, leads to a read_write capability. This is important for system blocks that must be protected but that lead to writable blocks in a user's space.

IV. SEGMENT-TABLE OPERATIONS

CREATE SEGMENT

Format: length (I), pointer to base (P) |
pointer to base of new segment (P)

This operation creates a new entry in the segment table of the process issuing the operation and returns a pointer to it. It can only be used to create a data and program segment. The read/write capability of the returned pointer is the same as that in the supplied pointer. If the pointer value supplied is not NULL, the newly created segment is a pseudo-segment (it can only be created on a data and program segment). When the supplied pointer is NULL, the new segment is a real segment for which core has not yet been allocated. A CORE NOT YET ALLOCATED entry is made, the process number of the retriever (see p. 16) is set to the monitor of the process that issued the operation, and the DATA AND PROGRAM UNMODIFIED segment-type is put into the SAVED SEGMENT TYPE field in the second word of the entry. When the segment is actually accessed, a SEGMENT FAULT occurs and the process' monitor must allocate core for the segment before the process may continue.

DESTROY SEGMENT

Format: pointer to segment (PW) |

The supplied pointer must have write capability. It cannot destroy a segment table or stack segment. The indicated segment is freed. Any disc space (see p. 16) allocated to the segment is freed. If the segment has real core allocated to it at the time the destroy is issued (i.e., if it is a real segment), a DESTROYED SEGMENT interrupt is generated for the process' monitor.

GIVE SEGMENT

Format: process number of receiver (I), receiving segment number (I), pointer to segment to be given (PW) |

The process specified (Process B) must be an immediate subprocess (see p. 18) of the process that issued the operation (Process A). If the segment being GIVEN is a SEGMENT ON SECONDARY STORAGE or a CORE NOT YET ALLOCATED entry, a SEGMENT FAULT occurs and, as described earlier, processing of the issuing process (Process A) is suspended until the segment becomes a REAL SEGMENT. If the receiving segment number specifies a FREE SEGMENT, a REAL SEGMENT, a PSEUDO SEGMENT, or a REMOTE SEGMENT to a process that is not an immediate subprocess of Process B, or if the segment being GIVEN is a FREE SEGMENT, a REMOTE SEGMENT, a PSEUDO SEGMENT, or a REAL SEGMENT that is not a DATA AND PROGRAM segment, then an INVALID GIVE program error occurs. If the receiving segment number specifies a REMOTE SEGMENT to an immediate subprocess of Process B, that process and segment become the receivers of the operation and the new receiving segment is processed as described above.

Ultimately, either an INVALID GIVE occurs, which halts the operation, or a SEGMENT NOT IN CORE entry is found. In the latter case, the ultimate receiving segment is read into the real core if it is a SEGMENT ON SECONDARY STORAGE. Any processes waiting for the segment (i.e., any that are P'ed on the semaphore in the segment-table entry) are V'ed. The entry is changed to a REAL SEGMENT representing the real core in the segment being given, the SEGMENT TYPE is set to the SAVED SEGMENT TYPE, and the segment being GIVEN is changed to a REMOTE SEGMENT to the original receiving segment and process.

RETRIEVE SEGMENT

Format: pointer to segment (P) |

The segment specified must be a REMOTE SEGMENT to an immediate subprocess (Process B) of the process (Process A) issuing the operation. An INVALID RETRIEVE program error occurs if the segment being retrieved is a PSEUDO SEGMENT, a REMOTE SEGMENT to a process that is not an immediate subprocess of Process B, a FREE, or a CORE NOT YET ALLOCATED segment. If the segment being retrieved is a REMOTE SEGMENT to an immediate subprocess of Process B, then that segment becomes the operand of the operation and the segment being retrieved is processed as described above.

Ultimately, either (1) an INVALID RETRIEVE must occur that halts the operation, or (2) either a REAL SEGMENT or a SEGMENT ON SECONDARY STORAGE entry must be found. If a modified REAL SEGMENT is found, it is copied to the location specified by the file pointer in the second word of this entry. If the file pointer is null, before the copy operation is performed, an area of secondary storage of sufficient length is allocated and a file pointer to it is placed in the second word of the entry. After the copy operation, if necessary, the initial REMOTE SEGMENT in Process A is changed to a REAL SEGMENT representing the real core; the segment type is picked up from the SAVED SEGMENT TYPE field of the second word of the entry; and the retrieved segment is changed to a SEGMENT ON SECONDARY STORAGE entry, its SEMAPHORE PROCESS NUMBER and SEMAPHORE COUNT are set to zero, and its PROCESS NUMBER OF RETRIEVER is set to the process number of the process that issued the operation (Process A).

If a SEGMENT ON SECONDARY STORAGE entry is found, some process for which Process A is a subprocess has already retrieved the real core associated with the segment. The initial remote segment in Process A is changed to a SEGMENT ON SECONDARY STORAGE entry; its SEMAPHORE PROCESS NUMBER and SEMAPHORE COUNT are set to zero; its PROCESS NUMBER OF RETRIEVER is set to the PROCESS NUMBER OF RETRIEVER in the entry found; the PROCESS NUMBER OF RETRIEVER in the entry found is set to the process number of the process that issued the operation (Process A); and, if any processes are waiting on the entry found, a SEGMENT REQUESTED interrupt is generated for Process A (which informs it that the retrieved segment has been requested).

MAKE PROCESS INACTIVE

Format: process number |

The indicated process must be an immediate subprocess of the process that issued the operation. All the real core given to the subprocess is retrieved and the segment tables for that subprocess and any of its subprocesses are retrieved by the ISPL machine, forcing them to be saved on secondary storage. The FILE POINTER to the location of the segment table replaces the value of the SEGMENT TABLE POINTER in the PCB.

MAKE PROCESS ACTIVE

Format: process number |

The indicated process must be an immediate subprocess of the process (Process A) that issued the operation. The segment tables for that process and any of its subprocesses that were made inactive by Process A are brought back into core in segments GIVEN by the ISPL machine. The size for each segment table is picked up from the PCB (in the MAX_SEGMENT_SIZE_USED field). Finally, the absolute address of the segment table is placed in the SEGMENT TABLE POINTER field of the PCB.

PROCESS MONITOR

Format: process number | process number of monitor

The PROCESS MONITOR operation returns the process number of the monitor of the specified process.

V. MULTITASKING, SCHEDULING, PORTS, AND INTERRUPTS

In the ISPL machine, multitasking, scheduling, Ports, and interrupts are intimately related. They are the mechanisms that coordinate and control asynchronous processes. As explained earlier, a process is one or more logical lines of control flow in a common address space. It is represented in the ISPL machine by a PCB that contains the "machine context" of the process--all the information needed to properly resume execution of that process. A PCB is created by a form of the INITIATE operation, which sets up a separate address space. This new process is controlled by, and is an immediate subprocess of, its monitor--the process that issued the INITIATE operation. This dependence is recorded in the PCB so that the monitor for any process is available; hence, if a subprocess initiates further subprocesses and acts as their monitor, the ISPL machine maintains the process hierarchy.

If a monitor has INITIATED several processes, it must allocate execution processing among them. This allocation is called scheduling. The decision is effected through the DISPATCH command, which is the process-equivalent of a subroutine call. The process that issued the DISPATCH operation is suspended, its machine context saved, the fact that it DISPATCHED another process recorded, and execution of the DISPATCHED process resumed as indicated by the machine context. This process continues to execute until it either asks for an unavailable resource or an interrupt occurs (such as the expiration of a timer set by the dispatching process). Process completion is treated as an interrupt.

A request for an unavailable resource is treated as a normal return from the dispatched process. This process' machine context is saved and the dispatching process is resumed at the syllable after the DISPATCH.

The second case concerns interrupts. An interrupt is the sudden availability of a resource required by a process. Resources in the ISPL machine are represented by a primitive data-type called a semaphore. Only certain operations are allowed with semaphores. These include the V operation, which makes a resource available (i.e., returns it); the P operation, which requests or obtains a resource if it is available and, if it is not, waits until it is available; and the CONDITIONAL P operation, which performs a P operation only if the resource is available.

Thus, an interrupt is a V operation on a semaphore for which some process has an active P operation--i.e., one that has not yet received the requested resource. Certain interrupts, such as process completion, segment fault, and all program errors, are generated by the ISPL machine upon recognition of the condition indicated. Other resources are represented by user-created semaphores; interrupts occur when these semaphores are V'ed if there is an active P on the semaphore.

The representation of and operations on all semaphores are the same whether they are machine- or user-created. However, the method of accessing the appropriate semaphore differs. User-created semaphores are always accessed by the address of the semaphore. For efficiency, and because both the ISPL machine and programs written in ISPL must be able to access the machine-defined semaphores, these semaphores are accessed by a machine-designated number. The MACHINE SEMAPHORE PRESENCE field indicates the presence or absence of the associated machine-defined semaphore. A P operation on such a semaphore is always treated as the creation of a new semaphore to handle the condition. The new semaphore is added to the top of the STACK of machine-defined semaphores present, and the MACHINE SEMAPHORE PRESENCE field is set to indicate the semaphore's presence. Because the semaphore is new, it will never be available when requested and will cause the executing unit to wait for the condition. A V operation on a machine-defined semaphore tests for its presence in the running process. If it is there, the semaphore stack is searched to find it. Since P operations always create a new instance of the semaphore, which is put on the top of the semaphore stack, the most recent semaphore is used if there are multiple active Ps on the same machine-defined semaphore. This is precisely the interpretation necessary for ON-UNITS in ISPL. If the machine-defined semaphore is not present in the running process, that process' monitor is checked for the semaphore's presence. This is repeated until a process that will handle the machine-defined semaphore is found. If no process is found, the ISPL machine must handle the semaphore as a system error.

To handle ON-UNITS, co-routines, and independent asynchronous tasks that share a common addressing space, the control blocks for a process are divided into three separate blocks:

1. The PCB, which holds all the common information about the process as a whole;
2. The INDEPENDENT EXECUTION BLOCK (IEB), which represents a single, independent, asynchronous, logical flow of control in the address space of the process;
3. The EXCLUSIVE EXECUTION BLOCK (EEB). One EEB exists for each of the ON-UNITS and one for the main line of the IEB. These EEBs contain the information local to the separate logical lines of control within the IEB. Only one EEB can logically be executing at once.

Each control block has the following fields:

PCB

IEB CHAIN (I)	CURRENT IEB (I)
PROCESS MONITOR (I)	INTERRUPT PENDING IEB (I)
SEGMENT-TABLE POINTER (P)	
IEB SWITCHING ENABLED (I)	ADDRESS BOUNDARY REGISTER (B)
SEGMENT-TABLE SIZE IN DOUBLE WORDS (I)	MADE INACTIVE BY (I)
RESERVED FOR FUTURE USE	

EEB

EEB CHAIN (I)	IEB NUMBER (I)	
DISPATCHED PCB (I)	EEB PRIORITY (I)	ENABLED (I)
SEMAPHORE WAIT CHAIN (I OR P)		
TOP OF STACK POINTER (P)		
INTERPRETATION POINTER (P)		
SYLLABLE POINTER (P)		

IEB

IEB CHAIN (I)		CURRENT EEB (I)	
EEB CHAIN (I)		INTERRUPT PENDING EEB (I)	
CREATING IEB (I)		IEB PRIORITY (I)	ENABLED (I)
EEB SWITCHING ENABLED (I)	IEB DISPATCH-ABLE (I)	RESERVED FOR FUTURE USE	
MACHINE-SEMAPHORE PRESENCE (B)			
RESERVED FOR FUTURE USE			

EEBs represent separate flows of control within an IEB; only one can logically be operating at once. If an ON-UNIT waits for a resource, the interrupted process cannot continue; it may only be resumed when the ON-UNIT is completed. This exclusive property and the sufficiency of a simple number to indicate the relative priority of EEBs enable scheduling within an IEB to be performed as an ISPL-machine function. Similarly, the sufficiency of a simple numerical priority among IEBs--representing separate independent logical flows of control (TASKS) within a process--many of which can logically be executing at once, enables scheduling of IEBs to be performed as an ISPL-machine function. On the other hand, interprocess scheduling is a monitor function because it may involve many variable factors. A monitor need only concern itself with scheduling its immediate subprocesses. The internal behavior of these subprocesses--whether they are in the main line, an ON-UNIT, or have dispatched one of their own subprocesses--is handled by the ISPL machine.

When an interrupt occurs, an EEB that was waiting for a resource is now able to execute. If it (1) is enabled and has a higher priority than the current EEB of that TASK (IEB) in the process (the one which will be in execution when the TASK is the current IEB of the process and the process is DISPATCHED) and (2) has a higher priority than any INTERRUPT PENDING EEB, then the higher-priority EEB is marked as the INTERRUPT PENDING EEB of the TASK.

Although interrupts can occur (a semaphore can be V'ed) at any time, they are only honored between source statements in that TASK's current EEB. Thus, at the point an interrupt is honored, the state of both the ISPL machine and the source program is well defined. The postfix translation of a program contains NEW STATEMENT operators, which define these points of interruptability. (The NEW STATEMENT operator also helps maintain source-object correspondence and facilitates incremental program editing.) An EEB also becomes interruptable when a program error, a P operation that causes a wait, or a DISPATCH operation occurs. This momentary interruptability is recorded in the high-order byte of the SYLLABLE POINTER field in the EEB.

When an interrupt occurs, if the TASK's current EEB is interruptable and the TASK does not have all interrupts masked, the INTERRUPT PENDING EEB becomes the CURRENT EEB and is reset to zero. Only if this EEB switch makes the TASK dispatchable (when it was not dispatchable before the switch) is the above process repeated to see if a TASK switch should occur. The dispatchability of a TASK is determined by the SEMAPHORE WAIT CHAIN field of the TASK's CURRENT EEB. If this field is non-zero, the EEB is waiting for a resource and the TASK is nondispatchable. The TASK's dispatchability is marked in the IEB DISPATCHABLE field. If the TASK was already dispatchable, no further interrupt processing is necessary. The next time the affected TASK is the current TASK of the dispatched process, the switched EEB in the TASK will be the one resumed. If the TASK was not dispatchable before the EEB switch, and if it is enabled and has a higher priority than the current IEB and any INTERRUPT PENDING IEB, then the TASK becomes the INTERRUPT PENDING IEB. If the process has TASK-switching enabled, the INTERRUPT PENDING IEB becomes the CURRENT IEB and is the TASK dispatched when the process is next DISPATCHED.

If the process became dispatchable (i.e., the CURRENT EEB in the CURRENT IEB is not waiting) only as a result of the interrupt, a PROCESS DISPATCHABLE interrupt is generated for the process' monitor to inform it that one of its subprocesses is now ready to be DISPATCHED. If the process was already dispatchable, there is no need to inform its monitor that an interrupt has occurred within it since the ISPL machine handles the intraprocess scheduling and the process' external state (its dispatchability) has not changed.

A distinction exists between (1) an EEB that is waiting for an asynchronous interrupt, such as I/O complete, and that will be V'ed by some other process, and (2) an EEB that is waiting for a synchronous interrupt, such as zero divide, and that will be V'ed either directly by another EEB in the same process or as a result of some action of that EEB (such as dividing by zero or end-of-file reached). The ISPL machine must handle these cases very differently. In the case of an EEB waiting for an asynchronous interrupt, the EEB remains as the CURRENT EEB of the process. Because that EEB is not dispatchable, the ISPL machine saves its context, indicates its interruptability, and resumes the dispatching process at the syllable following the DISPATCH operation. When the asynchronous interrupt occurs (assuming that no higher-priority EEB in that process has since become dispatchable), the process' monitor is given a PROCESS DISPATCHABLE interrupt; when the process is next dispatched, the suspended EEB is resumed.

In the case of synchronous interrupts, the interrupt (by definition) cannot occur until some other EEB within the process is resumed. Therefore, waiting for a synchronous interrupt (a P operation on a synchronous semaphore) is interpreted as both a P operation and a return from the EEB. Until the EEB again becomes dispatchable, the ISPL machine schedules lower-priority EEBs in the same TASK.

The distinction between synchronous and asynchronous semaphores is made in the ISPL machine by the SEMAPHORE TYPE field within the semaphore. This field is set when the semaphore is created. It indicates not only the synchronous-asynchronous status, but also whether or not the semaphore has any associated data. Some semaphores are used only to synchronize a set of processes, to signal an event, or to obtain exclusive access to a nonsharable resource. Other semaphores not only perform these functions, but also make available an item of data when the P operation is successfully completed. The meaning of this data is established, as is the meaning of the semaphore itself, by convention between the processes or EEBs that P and V the semaphore. For example, it might indicate where a zero divide occurred, which disc track is available, or which process became dispatchable. Three separate methods are provided for holding the data. UNBUFFERED semaphores have the data stored immediately following the semaphore. Such semaphores must be used so that no more than one data item is simultaneously stored with the semaphore (i.e., the count is never greater than one). The other two methods for holding the data make use of the STACK and QUEUE list processing structures[†] to provide, respectively, last-in-first-out and first-in-first-out buffering of the data.

Operations on data semaphores are somewhat different from nondata operations. When a V operation is performed, the data to be associated with the semaphore must also be supplied, and when a P operation is performed, the address of a variable to which the obtained data is to be assigned must be supplied.

One particular use of data semaphores bears special attention. PORTS, which are a central concept in the ISPL machine, are composed from data semaphores. PORTS are a program's or process' method of communicating with the outside world and of establishing co-routine linkage both within and between processes. A PORT is one end of a two-way communication path. The CONNECT machine-operation establishes this path between two PORTS. The PORT is a primitive ISPL-machine Plex; it is composed of a pointer to the PORT on the other end of the communication path (i.e., the connected or remote PORT) and a data semaphore for which the data is a pointer to a parameter_list (technically, an array of argument descriptors) that represents the data logically being passed through the PORT. The data may be stored in any of the allowed methods for data semaphores, UNBUFFERED or buffered in a STACK or QUEUE. A PORT is like a subroutine CALL in that (1) any number and type of arguments can be passed at once, and (2) the meaning of these arguments is established by convention between the caller (sender) and callee (receiver). PORTS and subroutine calls are also alike in that they both physically involve the passing of a single pointer to the parameter list.

PORTS and subroutine calls differ greatly in the way they affect control flow. Subroutine calls are always synchronous. The calling program is suspended; the called program is started (not resumed) at

---

[†]See Ref. 1, pp. 16-17.

its entry point (i.e., its PROCEDURE statement), runs to completion, and returns to the calling program, which then resumes execution.[†] PORTS involve a much more flexible control flow. Sending data through a PORT is simply a V operation on the remote PORT's data semaphore, supplying the parameter list pointer as data. Thus, the SEND operation merely makes the data available. Control-flow is determined by the rules of semaphore synchronization (see pp. 21-22). Briefly, if the V'ed semaphore does not have an active P, the sending program continues. If the V'ed semaphore has an active P from an EEB in the same process as the sending EEB, and if it has a lower priority or is not enabled, the sending EEB continues. If the V'ed EEB has a higher priority than both the sending EEB and the INTERRUPT PENDING EEB, it becomes the INTERRUPT PENDING EEB. The next point of interruptability (NEW STATEMENT operator) at which interrupts are enabled for the process causes an EEB switch within the process by the ISPL machine. Finally, if the V'ed semaphore has an active P from an EEB in a process different from the sending EEB, the same rules determine whether an EEB switch should occur within that process. If that process' CURRENT EEB is nondispatchable (i.e., waiting), the EEB switch occurs immediately. (Execution is not resumed in the process, its CURRENT EEB and INTERRUPT PENDING EEB are merely updated.) If a previously undispatchable process is now dispatchable--either as a result of an EEB switch or because the CURRENT EEB was the one that was P'ed on the V'ed semaphore--the ISPL machine generates a PROCESS DISPATCHABLE interrupt for that process' monitor to inform it of the change in status. Control-flow between these and other processes is the responsibility of the process' monitors.

Similarly, receiving data through a PORT is simply a P operation on the PORT's data semaphore, followed by the assignment of the pointer as the CURRENT instance of the parameter list being received. This makes the passed arguments accessible to the receiving program. Again,

---

[†]Systems that have co-routines also employ the subroutine call and return mechanism for this purpose. We feel that this facility is more naturally a part of the semaphore synchronization process. It is therefore a particular use of PORTS as explained below.

control-flow is determined by the availability of the semaphore and the rules of semapho.e synchronization.

PORTS always involve a form of co-routine linkage[†] because (1) they pass information back and forth between EEBs in the same or different processes; (2) EEBs maintain the machine context of a logical flow of control in a process; and (3) when the EEB is resumed, control continues at the SYLLABLE following the SYLLABLE at which the EEB was suspended. Whether the co-routines are synchronous or not depends on whether the sending and receiving EEBs are in the same process, what the relative priorities between them are if they are in the same process, and what the scheduling algorithms of their monitor or monitors are if they are not in the same process.

[†]As originally defined by Conway [2].

REFERENCES

1. Balzer, R. M., *The ISPL Language Specifications*, The Rand Corporation, R-563-ARPA, August 1971

2. Conway, M., "Design of a Separable Transition-Diagram Compiler," *Communications of the ACM*, Vol. 6, No. 7, July 1963, pp. 396-398.