AD731657

R-605-ARPA

August 1971

Ports—A Method for Dynamic Interprogram Communication and Job Control

R. M. Balzer

A Report prepared for

ADVANCED RESEARCH PROJECTS AGENCY

D C
RECEIVED
OCT 26 1971
B

Rand
SANTA MONICA, CA. 90406

DOCUMENT CONTROL DATA

1. ORIGINATING ACTIVITY	2a. REPORT SECURITY CLASSIFICATION
The Rand Corporation	UNCLASSIFIED
	2b. GROUP

3. REPORT TITLE

PORTS--A METHOD FOR DYNAMIC INTERPROGRAM COMMUNICATION AND JOB CONTROL

4. AUTHOR(S) (Last name, first name, initial)

Balzer, R. M.

5. REPORT DATE	6a. TOTAL NO. OF PAGES	6b. NO. OF REFS.
August 1971	24	-

7. CONTRACT OR GRANT NO.	8. ORIGINATOR'S REPORT NO.
DAHC15 67 C 0141	R-605-ARPA

9a. AVAILABILITY/LIMITATION NOTICES	9b. SPONSORING AGENCY
DDC-A	Advanced Research Projects Agency

10. ABSTRACT

Describes Ports, a unified method for communication between a computer program and terminals, files, peripheral devices, other programs, and supervisory software. In ISPL (Incremental System Programming Language, described in R-563), each job has a Port named MONITOR that handles resource allocation: creating and deleting files, assigning file space, core space, processor time. This design permits a hierarchical system of monitors, each controlling the jobs running under it. By routing output to a user terminal, Ports enable on-line debugging and simulation of rewritten files of programs. The Port concept improves modularity in 3 ways: (1) Each connection need not be specified by the programmer but can be decided at execution. (2) Linkage between programs is co-routine rather than subroutine, which simplifies programming, retains context, and removes the need for hierarchical organization; (3) With different connections via Ports, the same system can be used in many ways, e.g., on-line or off, in simulation mode, audit-trailed, or data breakpointed.

11. KEY WORDS

Computer Programming
File Structure and Management
Computer Simulation
ISPL

Ports—A Method for Dynamic Interprogram Communication and Job Control

R. M. Balzer

A Report prepared for

ADVANCED RESEARCH PROJECTS AGENCY

PREFACE

This report describes a unified method for communication between a computer program and files, terminals, physical devices, other programs, and the supervisor. The report (1) defines this method and its implementation; (2) describes and evaluates its uses for job control, debugging, simulation, and (perhaps most importantly) modularity; and (3) presents practical examples.

This study is part of the ARPA-sponsored research to improve man-machine interaction. It should be of interest to those concerned with a proper programming environment for research and development applications.

SUMMARY

This report presents a unified method for communication between a computer program and files, terminals, physical devices, other programs, and the supervisor. The method consists of a pair of interconnected Ports, each composed of a pointer to the other and a data semaphore (which allows data to be associated with a semaphore and buffers such data).

Information is passed through and obtained from a Port by the SEND and RECEIVE commands, respectively. The actual data passed is a pointer to a parameter list. This allows the same mechanism to be used as for subroutine arguments and facilitates the use of Ports for co-routine linkage. The CONNECT command is used to interconnect two Ports and can be issued by "supervisory" programs to fit a program into its operating environment, i.e., as a form of job control.

The facilities provided by Ports were obtained by combining into a single mechanism three powerful software techniques: co-routine, indirect specification, and communications commonality.

CONTENTS

I. INTRODUCTION

Without communication mechanisms, a program is useless. It can neither obtain data for processing nor make its results available. Thus, every programming language has contained communication mechanisms, which have traditionally been separated into five categories based on the entity with which the program communicates: (1) physical devices (printers, card readers, etc.), (2) terminals (although these are physical devices, they are usually treated separately), (3) files, (4) other programs, and (5) the monitor. One or more communication mechanisms correspond to each of these categories; some mechanisms may be shared between categories.

The "alphabet soup" in the example below indicates how diverse communication mechanisms have become. In IBM'S OS/360 [1], communication with physical devices is through either BSAM (Basic Sequential Access Method) or QSAM (Queued Sequential Access Method); terminals use BTAM (Basic Telecommunications Access Method), QTAM (Queued Telecommunications Access Method), or GAM (Graphics Access Method); files utilize BSAM, QSAM, BDAM (Basic Direct Access Method), BISAM (Basic Indexed Sequential Access Method), or QISAM (Queued Indexed Sequential Access Method); communication with other programs is through subroutine calls, and with the monitor through supervisor calls. There are ten different mechanisms for the five categories; each mechanism has different commands for using the communication mechanism.

We propose to show that Ports offer a single unified mechanism for communicating with any of the five entities. Besides simplifying communications, this unification allows the dynamic specification of the entity being communicated with at execution time. This delayed binding can be effectively used both to debug and build more flexible programs and to create modular programs that can be easily plugged

together to form systems. The remainder of this report defines Ports, explains their use, and attempts to justify the above claims.

II. EVOLUTION OF PORTS

The concept of Ports evolved from work on a somewhat mistitled study, "Dataless Programming" [2], which tried to develop a programming language that would enable representations for data structures to be selected after a program was completed rather than before it was begun. Selection of a representation after a program is written is much more appropriate because at that point the programmer knows exactly how the data is used; beforehand, he must predict actual usage. The different syntactic forms used in common programming languages for the different representations force the decision to be made at coding time. "Dataless Programming," by using a common syntactic form and extending the operations across all the representations, allows the decision to be delayed until after coding is completed. In addition to the chosen set of standard representations, the user can create his own representations by supplying the necessary manipulative routines for use by the compiler in accessing, updating, adding, deleting, or inserting an element from the representation, or obtaining the next or previous element.

Because "Dataless Programming" was never implemented as a system, we tried other ways to test its ideas. The key concept was the ability to invoke a routine, either standard or supplied by the programmer, whenever a data structure was used. Not desiring to write a compiler, we looked for a centralized mechanism that could be controlled to invoke the proper manipulative routines. Such a mechanism exists in IBM's OS/360 [3]--the Data Control Block (DCB) used for files. Whenever an action is required on the file (e.g., read or write), the address of the appropriate routine is obtained from the DCB. These addresses are placed in the DCB when the file is opened. The open process was modified so that, for selected files, the address of an interface

program, JOINER, was placed into the DCB rather than the address of a standard OS access method.

The JOINER program acted as an interface and controller between two DCBs that it had logically connected. Thus, the output of one program was available as input to another program. Each program acted as the access method for the other. For example, in Fig. 1, Program A has a DCB, called OUT, used for output that has been joined to a DCB, called IN, used for input to Program B.

```
                    ┌──────────┐
                    │  JOINER  │
                    └──────────┘
                      ↗      ↘
┌──────────────┬───┐          ┌───┬──────────────┐
│              │ o │          │ i │              │
│  PROGRAM A   │ u │          │ n │  PROGRAM B   │
│              │ t │          │   │              │
└──────────────┴───┘          └───┴──────────────┘
```
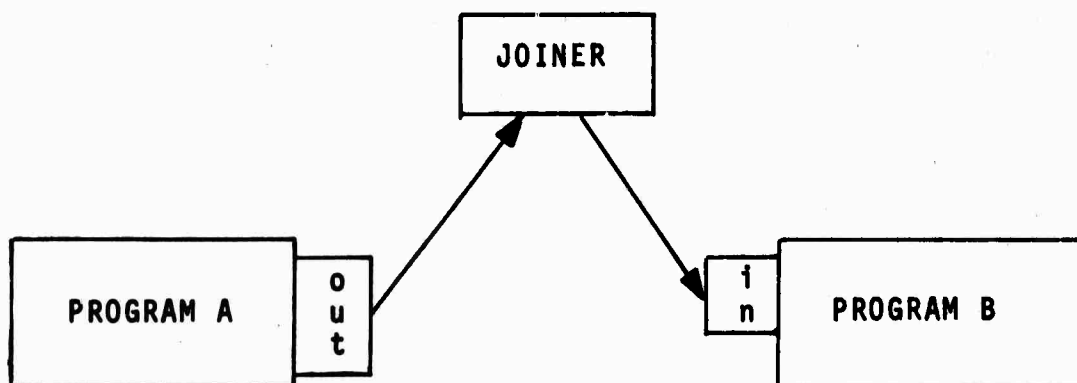
Fig. 1--JOINER Example

Assume JOINER has loaded Programs A and B, and has started A. Program A will open DCB OUT, and the address of JOINER will be placed in this DCB. Eventually, A will try some output through the OUT DCB, invoking JOINER. JOINER now starts B, and when B performs an input operation on its IN DCB, JOINER gives B the output from Program A. When B asks for the next input, JOINER suspends the program and restarts A to obtain more output to give B as input. JOINER thus coordinates the two programs and allows each to be used as the other's access method. Note that a type of co-routine

relationship is established between the programs [4]. This relationship is called Data-Directed Co-Routines because control is switched back and forth between the two programs as data is produced and required. The connection between the two programs exists outside of each of them, and they are unaware of what they are communicating with.

The JOINER system described above contains the key elements of Ports (defined in Sec. III). However, because it tested the ideas in "Dataless Programming," we needed to demonstrate some practical uses for this system.

We first added some macros to IBM's assembly language, which gave it a control-block structure. These macros are IF, ELSE, and ENDIF [5]. The IF macro begins a control block that is executed only if the condition tested by the macro is true. This control block is ended by either an ELSE or ENDIF macro. The ELSE macro ends the IF control block and starts an ELSE control block that is executed only if the condition tested by the IF macro is false. Because these macros can be nested, a noniterative control structure analogous to those of PL/1 or ALGOL is created. These macros are very heavily used and the nesting levels often extend ten levels and beyond. Hence, to make the program more readable, we built a formatting program that names the levels and indents the listing according to these levels.

Then, with JOINER, we connected the output of the assembler with the input of the format program. The connection is specified to JOINER and neither program is altered. Joining these two programs reduces (1) CPU and I/O charges, and (2) the elapsed time needed to run the job.

The second application of JOINER is even more important because it is the basis for an entire time-sharing system built under O/S. The Rand-built system is called Simultaneous Graphics System (SGS).[†] When a job is to be started, SGS

[†]SGS is an internal Rand time-sharing system.

joins the input of an O/S reader to the output of a spool
program. The spool program is necessary because the source
files are kept on the disc in compressed form as a linked
list so that they can be very rapidly updated. The spool
program follows the linked list and converts the file to the
required sequential set of 80-character card images. When
the job is running and requires input from or output for
the SGS file system, its DCBs are joined with the spool pro-
gram to provide the needed conversions. In this way, we are
able to run unmodified, standard OS/360 programs that utilize
the SGS file system, including such IBM processors as the
PL/1 compiler and the assembler.

III. DEFINITION AND IMPLEMENTATION

As presented in Sec. II, Ports can be defined as a data element used for communication with files, terminals, physical devices, other programs, and the monitor. Four basic operations can be performed on Ports. A Port can be CONNECTed to or DISCONNECTed from another Port, and data can be sent (SENDed) or RECEIVEd through a Port. REQUEST, a compound operation consisting of a SEND followed by a RECEIVE, is used for requesting certain data. The reverse sequence, a RECEIVE followed by a SEND, used for replying to a REQUEST, does not exist as a single operation because an arbitrary amount of processing may be needed between the RECEIVE and the answering SEND.

This definition, although containing the essence of Ports, does not answer many questions about Ports and their operation. For example, we need to know how data is passed through a Port; when control is transferred to the co-routine; what happens if two SENDs occur before the co-routine processes the first one; if two Ports can be connected to a third; and how Ports are connected to a terminal, physical device, or file. Ports can be logically implemented in different ways; each way might provide different answers to such questions. Each way is a logical implementation--one that produces logically different behavior as a result of the operations. We describe Ports in terms of one such logical implementation, ISPL [6-7], rather than JOINER, in which we are severely limited by the environment.

Incremental System Programming Language (ISPL) is both a language and an environment for programming. The ISPL language is an incrementally compiled PL/1-like language designed to run on the ISPL machine, which is designed specifically to run programs written in the ISPL language and is intended for implementation through microcode. As of this writing, the ISPL system is being implemented by a Rand

development team. All further discussion of Ports is in terms of this logical implementation.

In this implementation, Ports are defined in terms of "data semaphores," an extension we have made to Dijkstra's semaphores [8] that allows data to be associated with such semaphores. We have extended his definition as follows (the extensions are in italics):

Semaphores are a basic language data type used for synchronization. A semaphore logically consists of a count of the available resources of a particular type. The only legal operations on a semaphore are the P, V, and *conditional P* operations. The P operations request one resource. The semaphore's count is decremented by one, and if the result is nonnegative, the requestor continues. Otherwise, the requestor must wait until the resource is made available. The V operation makes a resource available. It increments the semaphore's count by one and if the result is nonpositive, one of the waiting requestors is reactivated. *The conditional P operation performs a P operation only if the requested resource is available, and returns an indication of whether the resource was obtained or not.*

Semaphores may, in addition, have a datum associated with the available resource. Such semaphores are called data semaphores, and the legal operations for these semaphores are P data, V data, and conditional P data, which are like their nondata counterparts except that the V-data operation must also supply the data to be associated with the available resources, and the P-data and conditional P-data operations must specify a variable to which the data associated with the requested resource will be assigned. The data can be any item in the language to which the assignment operator applies, or a structure of such items. The data can be buffered in a stack or a queue, providing respectively, LIFO and FIFO availability. They may also be stored unbuffered for those data semaphores whose count is never greater than one.

Using the definition for data semaphores, we define Ports as a basic language data-type used for communication.

They consist logically of a pointer to the Port to which the connection is made and a data semaphore representing both the availability of and the actual data being passed through the Port. The only legal operations on Ports are CONNECT, DISCONNECT, SEND, RECEIVE, conditional RECEIVE, and REQUEST.

Because Ports are used for a type of co-routine call, the same mechanism used for transmitting data to a subroutine should be used for Ports. Thus, the data physically passed through the Port (and its data semaphore) is a pointer to an actual parameter list, the contents of which are accessed by the receiver through a formal parameter list. As with subroutines, a convention between the communicating programs establishes the data logically passed through a Port and its interpretation.

The CONNECT command interconnects two Ports by setting their pointers to reference each other. DISCONNECT sets the two pointers to NULL. When two Ports are connected, the Port specified in a SEND, RECEIVE, or REQUEST command is referred to as the local Port and the Port it is connected to is referred to as the remote Port.

The SEND command builds an actual parameter list from the data specified in the command and performs a V-data operation on the remote Port's data semaphore, with a pointer to the actual parameter list as the data. The data in the actual parameter list is now available to be received through the remote Port. The RECEIVE command performs a P-data operation on the local Port's data semaphore, specifying an internal cell to which the parameter-list pointer will be assigned and that will be used by the language's standard mechanism for accessing formal parameters. If no data is available, the requestor is suspended until it is available. The conditional RECEIVE is similar, except that a conditional P operation is used. The REQUEST command is simply a SEND followed by an unconditional RECEIVE.

So far, we have described the operations on Ports in situations where two Ports are interconnected, but have not handled the cases where a Port is connected to a terminal, physical device, or file. Terminals and physical devices are handled by connecting the Port to a Port in the appropriate device-dependent system program, which transforms the communication into I/O commands appropriate for the device and then requests the supervisor to perform the I/O through the MONITOR Port (see Sec. IV).

Files are handled similarly, except that the type of file specified determines the program to which the connection should be made. The ISPL file system [9] is based on the "Dataless Programming" principle that representation-extension capabilities should be provided by allowing the user to supply the manipulative routines necessary to implement the new representation. Thus, corresponding to each type of file, there exists a set of manipulation routines for creating, destroying, connecting, disconnecting, and communicating with files of that type. When the CONNECT command is issued, the file name is found in the master directory and its file type is used to access and execute the connect routine and to access the communication routine connected to the specified Port. Thus, Ports are always connected to other Ports. For terminals, physical devices, and files, the remotely connected Port is in a program selected by the system on the basis of terminal, physical device, or file characteristics.

We have answered the questions on detailed Port behavior posed in this section, except for specifying when control is transferred to the co-routine. To provide the required flexibility, ISPL's control structure is necessarily complex. Scheduling decisions are made at three levels: process, task, and exclusive-execution block. In ISPL, a process is a set of independent tasks that share a separate, unique, addressing space. It roughly corresponds

to a job. Processes are scheduled by their supervisors,
which are informed via an interrupt when one of their pro-
cesses waiting for some resource is again able to run.
Nothing more can be said about process scheduling because
each supervisor can use its own arbitrary scheduling
algorithm.

The ISPL machine controls all scheduling within a pro-
cess. Each task within a process is a logically independ-
ent flow of control that could be executed simultaneously
with other tasks if multiprocessors were available. Each
task has a relative priority, and the ISPL machine sched-
ules the task with the highest relative priority that is
not waiting.

Tasks, in turn, are composed of exclusive-execution
blocks, which are separate flows of control; even in a
multiprocessor system, only one exclusive-execution block
can logically be executing at a time. As with tasks, the
ISPL machine schedules exclusive-execution blocks within a
task on the basis of their relative priority among those
not waiting. The important difference between the two is
that if an exclusive-execution block is interrupted by one
with a higher priority, it will not be resumed when the
higher-priority one waits for some resource, as is the case
for tasks, but must wait for the higher-priority exclusive-
execution block to exit. This control structure is required
for the implementation of co-routines and the on-units of
PL/1 [10]. An exit occurs when a program completes or per-
forms a P operation on a synchronous semaphore--one which
will not asynchronously be V'ed. Because it will not be
V'ed asynchronously, it must be an exit so that some other
exclusive-execution block in the task can cause it to be
V'ed. In ISPL, each semaphore and Port can be either syn-
chronous or asynchronous. Thus, the control flow resulting
from SEND and RECEIVE operations on Ports depends upon (1)
whether the remote Port is in the same process or the same

task, and (2) what its priority is relative to the executing exclusive-execution block. This structure enables us to build control structures ranging from completely asynchronous execution to those that switch control every time a SEND or RECEIVE is executed.

IV. USAGE

Obvio1sly, Ports can be used to communicate between programs. But the capability to externally specify the connection and the arbitrary nature of the program to which the connection is made enable the Port mechanism to be used for a variety of other purposes.

Since batch and multiprogrammed monitors, job control has traditionally been handled through a special language. This job-control language has two main functions, allocation of resources and fitting the job into an environment. Fitting the job into an environment consists of setting up the communication paths between the job and the files, terminals, physical devices, programs, and monitor with which it is to communicate. This is precisely what Ports are designed for; the CONNECT command specifies this function. In ISPL, each job has a Port named MONITOR, which is used for all communication with the job's monitor. Because any program can be connected to this Port, this design allows for a hierarchical system of monitors, each controlling the jobs running under it. Naturally, ISPL's hierarchical design relies on much more than the Port mechanism,[†] but Ports solved the system's communications requirements.

Communication with the monitor through a Port provides the mechanism for handling the other main function of job control, allocation of resources. The creation and deletion of files, allocation of file space, allocation of core space for the job, and specification of the central processor requirements are all transmitted to the supervisor through the MONITOR Port. The format of these specifications is a convention established by the supervisor.

Ports can also be used for debugging and simulation. Because output from a program can be routed to a terminal,

---

[†]See Ref. 7 for a full description.

and input obtained from the terminal, a user can dynamically
supply test data based on the program's performance. The
user can also simulate the behavior of part of the system
while observing and debugging the rest. A TEST program can
be written to implement data breakpoints; that is, whenever
the data transmitted through the Port to which the TEST pro-
gram is connected satisfies the test condition, a "break"
occurs and the user at a terminal is notified or a printout
occurs. The output of the TEST program is the same as its
input so that the TEST program does not affect the logical
processing of the program being debugged. A SPLITTER pro-
gram, whose two outputs are the same as its one input, can
be used to monitor, copy, or provide an audit trail of the
data transmitted through a Port.

The last two programs mentioned, TEST and SPLITTER,
offer examples of what we hope will be the major impact of
the Port concept--a mechanism for the construction of sys-
tems from small, general-purpose, "pluggable" programs.

Perhaps the single most important problem facing the
computer industry today is the inability to cheaply and
quickly generate debugged software systems. Many people
have proposed modularity as the solution, but such systems
have been hard to construct because of the strict hierarchi-
cal nature of subroutine calls--the only common method of
linking together such a set of programs.

The Port concept improves the construction of modular
systems in three important ways. First, the entity to which
the connection of a Port is made need not be specified with-
in that program; it can be dynamically decided at execution
time. Second, the linkage is co-routine rather than subrou-
tine, which simplifies the construction of many programs,
enables retention of context, and removes the strict hierar-
chical organization dictated by subroutine linkage. Finally,
connection of a Port can be made not only to Ports in other
programs, but also to terminals, files, and physical devices.

Thus, the same system can, with different connections, be used in a variety of ways--on-line, off-line, audit-trailed, data-breakpointed, or partial-user simulation.

The effectiveness of the Port concept results from the combination into a single mechanism of three powerful software techniques: co-routines, indirect specification, and communications commonality. We expect to extensively test the concept, especially its modularity potential, through its implementation in ISPL.

REFERENCES

1. *IBM System/360, Supervisor and Data Management Services,* Form C28-6646, IBM Corporation, Poughkeepsie, N. Y., 1967.

2. Balzer, R. M., *Dataless Programming,* The Rand Corporation, RM-5290-ARPA, July 1967. (Also published in *AFIPS Conference Proceedings,* Vol. 31, Thompson Books, Washington, D. C., 1967, pp. 535-545.)

3. *IBM System/360, System Control Blocks,* Form C28-6628, IBM Corporation, Poughkeepsie, N. Y., 1967, pp. 21-78.

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

5. Balzer, R. M., *Block Programming in OS/360 Assembly Code,* The Rand Corporation, P-3810, May 1968.

6. -----, *The ISPL Language Specifications,* The Rand Corporation, R-563-ARPA, August 1971.

7. -----, *The ISPL Machine: Principles of Operation,* The Rand Corporation, R-562-ARPA, August 1971.

8. Dijkstra, E. W., "The Structure of the 'THE' - Multiprogramming System," *Communications of the ACM,* Vol. 11, No. 5, May 1968, pp. 341-346.

9. Harslem, E., and J. Heafner, *The ISPL Basic File System and File Subsystem for Support of Computing Research,* The Rand Corporation, R-603-ARPA, August 1971.

10. *IBM System/360, PL/1 Reference Manual,* Form C28-8201, IBM Corporation, Poughkeepsie, N. Y., 1968.