Check for updates

PORTS—A method for dynamic interprogram communication and job control*

by R. M. BALZER

The RAND Corporation
Santa Monica, California

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. These mechanisms have traditionally been separated into five categories based on the entity with which communication is established. The five entities with which programs can communicate are physical devices (such as printers, card readers, etc.), terminals (although a physical device, they have usually been treated separately), files, other programs, and the monitor. Corresponding to each of these categories are one or more communication mechanisms, some of which may be shared with other categories.

The "alphabet soup" in the following example is used only to indicate how diverse communication mechanisms have become. In IBM's OS/360,[1] communication with physical devices is through either BSAM (**B**asic **S**equential **A**ccess **M**ethod) or QSAM (**Q**ueued **S**equential **A**ccess **M**ethod); terminals use BTAM (**B**asic **T**elecommunications **A**ccess **M**ethod), QTAM (**Q**ueued **T**elecommunications **A**ccess **M**ethod), or GAM (**G**raphics **A**ccess **M**ethod); files utilize BSAM, QSAM, BDAM (**B**asic **D**irect **A**ccess **M**ethod), BISAM (**B**asic **I**ndexed **S**equential **A**ccess **M**ethod), or QISAM (**Q**ueued **I**ndexed **S**equential **A**ccess **M**ethod); communication to other programs is through subroutine calls, and to the monitor through Supervisor Calls. There are ten different mechanisms for the five categories; each mechanism has different commands for the utilization of the communication mechanism.

We propose 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 utilized for both debugging and building more flexible programs, and as a means for creating modular programs that can be easily plugged together to form systems. The remainder of this Report is devoted to defining Ports, explaining their use, and justifying the above claims.

EVOLUTION OF PORTS

The concept of Ports evolved several years ago from work on a somewhat mistitled paper called "Dataless Programming."[2] In that effort, we tried to develop a programming language that would enable representation 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 are used; beforehand he must predict the 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 by 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 could create his own

* This study is part of RAND's ARPA sponsored research to improve man-machine interaction under contract DAHC 15-67-C-0141.

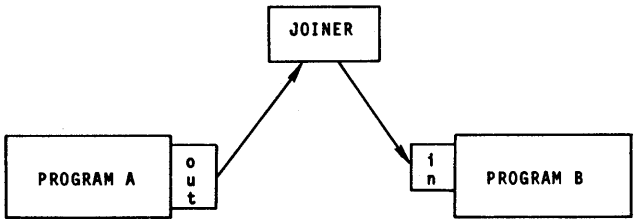

Figure 1—JOINER example

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 one.

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, such as read or write, the address of the appropriate routine is obtained from the DCB. These addresses are placed in the DCB at the time 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 together. Thus, the output of one program was available as input to another program. Each program acted as the access method for the other. Consider Figure 1. Program A has a DCB, called OUT, used for output that has been joined to a DCB, called IN, used for input in Program B.

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 the 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 access method for the other. Notice that a type of co-routine[4] relationship is established between the programs. This relationship is called Data-Directed Co-Routines because control is switched back and forth between the two programs as data are produced and required. Also note that the connection between the two programs exists outside of each of them, and that they are unaware of who they are communicating with.

The JOINER system described contains the key elements of Ports (defined in the next section). However, we need to demonstrate some practical uses for this system because it tests the ideas in "Dataless Programming."

We first add some macros to IBM's assembly language, which gives it a control block structure. These macros are IF, ELSE, and ENDIF.[5] The IF macro begins a control block that is executed if and 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 if and only if the condition tested by the IF macro is false. These macros can be nested, and hence a non-interactive control structure analogous to those of PL/1 or ALGOL is created. We find that these macros are very heavily used and that the nesting levels often extend ten levels and beyond. Hence, to make the program more readable, we build a formatting program that names the levels and indents the listing according to these levels.

Then, with JOINER, we connect the output of the assembler with the input of the format program. The connection is specified to JOINER and neither program is altered. By joining these two programs, we reduce both our CPU and I/O charges and the elapsed time needed to run the job.

The second application is even more important as 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 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 program 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.

* SGS is an internal RAND time-sharing system.

DEFINITION AND IMPLEMENTATION

As presented in the preceding sections, 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. They can be CONNECTed to or DISCONNECTed from another Port, and data can be sent (SENDed) or RECEIVEd through a Port. One compound operation, REQUEST, consisting of a SEND followed by a RECEIVE, and used for requesting certain data, also exists. The reverse sequence, 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 done between the RECEIVE and the answering SEND.

This definition, although containing the essence of Ports, does not answer many questions about Ports and the way they operate. We need to know how data are passed through a Port; when control is transferred to the co-routine; what happens if two SENDs occur before the first one is processed by the co-routine; if two Ports can be connected to a third; and how Ports are connected to a terminal, physical device, or file; etc. Ports can be logically implemented in many different ways, each providing different answers to the above and similar 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,[7,8] rather than JOINER, in which we are severely limited by the environment.

Incremental **S**ystem **P**rogramming **L**anguage (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 micro-code. 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] allowing 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 non-negative, 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 non-positive, 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 non-data 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 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, we feel 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, the data logically passed through a Port and its interpretation are established as a convention between the communicating programs.

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 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 which will be used by the language's standard mechanism for accessing formal parameters. If no data is available, then the requestor is suspended until one 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.

We have, so far, 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 a device-dependent system program for the terminal or physical device that transforms the communication into I/O commands appropriate for the device, and which then requests the supervisor to perform the I/O through the MONITOR Port (see the following section).

Files are handled similarly, except that the determination of the program to which the connection should be made is based on the type of file specified. 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 that is connected to the specified Port. Ports are thus 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 the characteristics of the terminal, physical device, or file.

The questions on detailed Port behavior posed in this section have now been answered except for specifying when control is transferred to the co-routine. To provide the flexibility we require, the control structure of ISPL is necessarily complex. Scheduling decisions are made at three levels. First is the process level. 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 that are informed via an interrupt when one of their processes, which is 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. All scheduling within a process is controlled by the ISPL machine. Each task within a process is a logically independent flow of control that could be executed simultaneously with other tasks if multi-processors were available. Each task has a relative priority, and the task with the largest relative priority that is not waiting is scheduled by the ISPL machine. Tasks, in turn, are composed of exclusive-execution blocks that are separate flows of control, but only one of which can logically be executing at once, even in a multi-processor system. 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 a higher priority one, 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-routine and the on-units of PL/1.[10] An exit occurs when a program completes or does 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 synchronous or asynchronous. Thus, the control flow resulting from SEND and RECEIVE operations on Ports depends upon whether the remote Port is in the same process or same task, and 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.

USAGE

Ports can obviously 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 utilized 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 func-

tions, 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 function is precisely what Ports are designed for, and is specified via the CONNECT command. In ISPL, each job has a Port named MONITOR, and it 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 (see Reference 7 for a full description), but Ports solved the communications requirements of the system.

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, the allocation of file space, the allocation of core space for the job, and the 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 purposes. Output from a program can be routed to a terminal, and input obtained from the terminal so that 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 program is connected satisfy 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 it does not affect the logical processing of the program being debugged. A SPLITTER program, 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 systems from small general-purpose "plugable" programs.

Perhaps the single most important problem facing the computer industry today is our inability to generate, cheaply and quickly, debugged software systems. Many people have proposed modularity as the solution, but such systems have been hard to construct because of the strict hierarchical 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 within that program, and can be dynamically decided at execution time. Second, the linkage is co-routine rather than subroutine. As others have suggested, this simplifies the construction of many programs, enables retention of context, and removes the strict hierarchical 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 R M BALZER
Dataless programming
AFIPS Conference Proceedings FJCC 1967 Vol. 31
Thompson Book Co. 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 M CONWAY
Design of a separable transition-diagram compiler
Communications of the ACM Vol 6 No 7 July 1963 pp 396–398

5 R M BALZER
Block programming in OS/360 assembly code
The RAND Corporation P-3810 May 1968

6 R M BALZER
The ISPL language specifications
The RAND Corporation R-563-ARPA (In process)

7 R M BALZER
ISPL Machine: Principles of operation
The RAND Corporation R-562-ARPA (In process)

8 E W DIJKSTRA
The structure of the 'THE'-multiprogramming system
Communications of the ACM Vol 11 No 5 May 1968 pp 341–346

9 E HARSLEM J HEAFNER
The ISPL basic file system and file subsystem for support of computing research
The RAND Corporation R-603-ARPA (In process)

10 *IBM System/360, PL/1 reference manual*
Form C28-8201 IBM Corporation Poughkeepsie N Y 1968