NASA TECHNICAL MEMORANDUM

NASA TM X-71908

NASA TM X-71908

(NASA-TM-X-71908) SOFTWARE HANDLERS FOR PROCESS INTERFACES (NASA) 15 p HC $3.50 CSCL 09B

N76-21932

Unclas
G3/61 21602

SOFTWARE HANDLERS FOR PROCESS INTERFACES

by Robert W. Bercaw
Lewis Research Center
Cleveland, Ohio 44135

TECHNICAL PAPER to be presented at
Symposium on Automatic Computation and Control
sponsored by the Institute of Electrical and Electronics
Engineers and Association for Computing Machinery
Milwaukee, Wisconsin, April 22-24, 1976

SOFTWARE HANDLERS FOR PROCESS INTERFACES

Robert W. Bercaw
Lewis Research Center

ABSTRACT

Process interfaces have been developed in an effort to reduce the time, effort, and money required to install computer systems. Probably the chief obstacle to the achievement of these goals lies in the problem of developing software handlers having the same degree of generality and modularity as the hardware. The problem of combining the advantages of modular instrumentation with those of modern multitask operating systems has not been completely solved, but there are a number of promising developments. This paper attempts to illustrate the essential principles involved.

E-8702

INTRODUCTION

Numerous modular interfacing systems have been developed to eliminate the custom interfacing problems in process I/O. These include both proprietary systems and non-proprietary systems such as the H-P ASCII bus (IEEE 488), the CAMAC Dataway (IEEE 583) and the CAMAC Serial Highway. In this paper, I would like to review the problems and some of the solutions involved in developing software handlers for such process interfaces. The discussion will be mostly in terms of the CAMAC standard, but most of the principles should be applicable to other systems.

The input-output structures of most modern software operating systems have been engineered primarily to provide efficient and flexible support of the standard computer peripherals such as disks, tapes, printers, etc. These devices are usually quite complex and carry price tags which are comparable to the other major components of the system. They are used intensely and their performance factors usually are significant in determining the overall performance of the total system. Consequently, it is standard practice to write a piece of highly specialized software, called a handler, for each peripheral installed in a system. The operating system is written in such a way that

these handlers can easily be added in a modular fashion.

Laboratory and process I/O devices usually are fairly simple, at least in regard to their interaction with the data system, and they normally have moderate use factors. It would therefore appear that the customary programming methods would be quite adequate to serve all of their needs. The problem lies in the large number of devices that may be required and in the extreme variability that occurs from job to job in both the number and types of devices that are required. The usual, handler per device, solution which works so well when there is only a small number of infrequently changed devices, becomes unwieldly and forces the writing of custom assembly language I/O software for individual applications. The problem is compounded by the presence of multiple hardware vendors and the "no man's land" status of the software support. In a sense, the problems which exist in the software are the same as the problems which prompted the development of process interfaces.

Because of the wide variety of applications for which process interfaces are used, it is impossible to precisely state the software requirements. Nevertheless, the following design goals are probably applicable to most situations:

1. The software should impose a minimal reduction in the generality of the process interface.

2. It should provide time responses and data transfer rates comparable to the speed of the hardware.

3. It should be compatible with multitask or other queued I/O operating systems.

4. It should require a minimum of programmer training and effort to use the handler in a particular application.

5. It should be accessible from Fortran.

6. It should possess minimal machine and interface dependence.

In addition, handlers for remote interfaces must be able to recover from transmission errors and allocate the use of the communications facilities amongst competing tasks.

SOFTWARE MODULARITY

The efforts of standards groups such as the Purdue Workshop have done much to clarify the nature of process I/O. There are basically two methods of organizing I/O through a process interface; by device or by address. The ISA-S61.1 standard FORTRAN calls are device oriented in that each call refers to functional I/O such as analog input, contact sensing, momentary contact closure, etc. An example is the subroutine which drives an analog digitizer in a random sequence:

CALL AIRD(LN, PAT, DATA, STATUS)

where: LN is the number of channels to be digitized, PAT is an array containing the physical address table, DATA is an array in which the data is placed, and STATUS is a word indicating the status of the transfer. The subprogram AIRD is expected to locate the digitizing module within the process interface and to issue the sequence of commands necessary to execute the call.

A more fundamental approach, based on the standardized interface structure, is taken by the CAMAC Software Working Group in defining their Fortran callable subroutine:

CALL CCSA(F, ADDR, DATA, STATUS)

transfers a single word between the integer variable DATA and whatever device is located at the interface address ADDR using the standard CAMAC function code specified by 'F'. A number of other subroutines are defined incorporating operational concepts such as block transfer, but the generation of any detailed device-dependent procedures is left to the applications programmer.

These approaches are not as different as it would first appear because the device oriented subroutines can be expanded in terms of the address oriented ones. For example:

```
      SUBROUTINE AIRD(LN, PAT, DATA, STATUS)
      COMMON/ADDR/
      INTEGER PAT(LN), DATA(LN), STATUS, ADDR
      STATUS = 2
      MSTAT = 0
      DO 1  I = 1, LN
      CALL CCSA(16, ADDR, PAT(I), LSTAT)
      MSTAT = MSTAT + LSTAT
      CALL CCSA(0, ADDR, DATA(I), LSTAT)
1     MSTAT = MSTAT + LSTAT
      STATUS = 1
      IF (MSTAT.NE.0) STATUS = 3 + MSTAT
      RETURN
      END
```

Similarly, it can be shown that all of the more complex calls can be reduced to a few fundamental calls such as CCSA. In principle, they are the only calls which have to be implemented by the handler and hence the handler can be made quite simple.

This is an example of a hierachical approach in which application programs talk to device-oriented routines, which in turn talk to the interface through address-oriented routines. It is very attractive because it mirrors the hardware hierarchy and provides the same modularity. Note in the example that the code specific to the ADC is in a computer independent language, Fortran. D. Zobrist (Ref. 1) illustrated the one-to-one correlation between hardware and software modules by the sandwich structure shown in Fig. 1 and pointed out that the software for each level can be written independently of the implementations of the other levels. Designers of computers, interface controllers, and modules can each write their own software without concern for the design details of the other components of the total system.

Problems occur, however, when one attempts to build a high performance I/O system within a multitasking environment. Most modern operating systems provide a multi-user multiprogramming environment in which the programmer can develop a program element or task with little concern for the operation of the other tasks. One of the ways in which the operating system protects its

integrity against programming errors is by executing all I/O itself. The user is prohibited from accessing the I/O structure and must post requests with the operating system for any I/O services. It generally queues up requests for a device and validates their correctness (e.g. it may require that any user supplied addresses lie within the users own space) before executing them. Because these processes typically take on the order of a milisecond to perform, the number of I/O requests which can be processed is limited to about 1000 per second and the response time of the system to an external event is drastically longer than the hardware response time. In the example of the multiplexed ADC, two I/O requests are processed for every word digitized and one second of CPU time is required to digitize 500 analog signals.

It is clear that the efficiency of the operation can only be significantly improved by eliminating the bulk of the I/O queuing and validating and that this is only possible by effecting block or multiword I/O from a single directive. But this means that the details of the transaction must be carried out by the handler and hence it must contain device-specific structures. The matching of hardware and software levels essential to software modularity is destroyed. The task of restoring modularity calls for a new approach and possibly a number of solutions for various performance ranges.

The principles of writing such generalized handlers are now emerging and can best be illustrated through some examples. Figure 2 summarizes the relevant I/O structure of a number of multitasking operating systems. The user's program issues a directive (or places a call) to execute an operation described by a specified Directive Parameter Block (DPB). The operation may be a READ, a WRITE, or some other operation, such as ATTACH in which the task gains exclusive use of a device. The operating system's queuing facility places the directive in a priority ordered gueue associated with the device's handler. The handler processes the directives sequentially. It firsts validates a directive, checking the DPB for correctness, and then dispatches it to a logical processor for execution. Finally, the directive node is removed from the queue and either the calling program and/or the operating system is notified of the I/O completion. The logical processors which actually execute the directives may be coded within the handler or they may be system subroutines which are called by it.

Because each directive is processed to completion before starting the next one, a directive could spend considerable time in the queue if it is preceded by others having long execution times. These waits are unnecessary and undesirable if the directives are for different physical devices and so most systems provide for multi-unit handlers which can support concurrent I/O to the different units. The I/O queuing facility maintains a separate queue for each physical unit and the logical processors are written in reentrant or reusable code with separate work areas set aside for each unit.

SOME HANDLERS

The first example is a handler for the CAMAC Serial Highway which was written to run under DEC's RSX-11D by Robert Setter at NASA (Ref. 2). It has subsequently been adapted to run under RSX-11M by the group at Inland Steel and the principles are being used at NASA for a handler to support a microcomputer-based remote acquisition unit. The handler has worked well and has proven to be quite reliable. It achieves better than an order of magnitude increase in throughput over simple handlers which use an I/O directive for each transfer.

The design is based on the conclusions of both the Purdue and CAMAC software groups that there is actually a relatively small number of useful I/O patterns in process control or other data acquisition. These can be built into the handler in the form of additional logical processors, each of which is coded to optimally execute one of the standard calls. Examples include the ISA random analog input call AIRD, the CAMAC single command execution CCSA, the Fortran WRITE to an alpha-numeric display or TTY, and the selective initiation of task execution by a contact closure. Each of these devices is assigned a block of several physical-address unit numbers so that concurrent I/O may be carried on with all of the modules which are installed on the system. In processing a request, the handler uses the unit number to look up the address of the correct logical processor in the Device Dispatch Table and to obtain device-dependent parameters such as its crate and station numbers and its buffer size.

In designing the handler, considerable effort was expended to make it modular in order to make it easy to add other devices as required. There is a rather clear separation between the portions that are relevant to the process I/O structure and those that are required for the integrity of the operating system. Although complete machine independence is not practical, the use of conditional assembly, separate parameter definition files and a computer independent I/O language such as IML would reduce the customizing of the handler to a cookbook process. The approach does have the disadvantage that the applications programmer must be able to modify and install a handler task (or perform the equivalent SYSGEN) and must be conversant with the particular computer's assembly language. An advantage is that it is possible to tailor the code to the peculiarities of the available hardware which all too frequently does not have the uniformity required for full software standardization.

An important step in providing a machine and application independent process interface handler has been made by the Real Time Systems Group at Lawrence Berkely Laboratory (Ref. 3). They have also placed device dependent code into the handler, but they have given the user the ability to create and destroy this code at run time. They have postulated that most or all I/O requests can be reduced to a series of primitive operations in areas such as buffer control, I/O status notification, and several data transfer patterns. The buffer control primitives include the setup of input and output buffers, and the chaining together of multiple buffers. The notification primitives conditionally return to the user or operating system the status of the various buffers, the number of CAMAC commands executed, the reasons for transaction termination, and other information concerning the status of the request. The data transfer primitives include (1) applying a list of random CAMAC commands, (2) applying a single function repeatedly to a sequence of addresses, and (3) applying a single function repeatedly to a single address. There are also primitives to provide conditional branching and hence permit the creation of sophisticated I/O programs.

The execution primitives are organized into the desired logical processor by a request structure which consists of a header, a work area, and a string of functional blocks. The header identifies the structure and its owner, while the work area provides space for the primitives to communicate with each

other and to hold information while the logical processor is inactive. Each of the functional blocks invokes one of the primitives and supplies it with the constants which control its operation. The above implementation of a logical processor is commonly referred to as "threaded code" and is similar in structure to Fortran object time systems of the same name.

I/O requests are implemented in two phases; an initialization phase in which the buffers, commands and other parameters are validated and the execution structure is built; and an execution phase(s) in which the actual transfer of data occurs. The real-time structure (or driver), shown in Fig. 3, consists of the above logical processors plus an initiation module which links each of them to the desired initiating condition, either an interrupt service routine (e.g. a LAM service) or a direct initiation from the I/O directive.

The initialization phase occurs some time before execution and consists of two logical parts invoked by two directives which may or may not be combined. The first creates the request structure while the second activates it by connecting it to an initiation condition. The CREATE directive passes a user-written parameter list to the handler, which first validates all of the vital parameters for possible fatal errors, and then uses them to build the functional blocks contained in the request structure. These are time consuming operations and have been placed in the pre-execution phase to streamline the real time transfer of data. Once the request structure has been created, it can be activated an arbitrary number of times by successive REQUEST I/O directives.

I believe that the most important aspect of the LBL handler is that it holds the key to several important developments in the area of machine independence and software portability. The functional blocks contain only process-specific parameters, not instruction set specific ones, and therefore it should be possible to develop a high-level I/O language and compiler to generate them. There seems to be no reason to exclude writing the compiler in Fortran or other machine independent language. It would thus be possible to actually achieve the goals postulated by the Zobrist Sandwich in which the software has a one-to-one relation to the hardware. The second implication of the principles of the LBL handler is that at least the driver portions (and perhaps some of the other parts) of the handler can be moved into an intelligent I/O processor, thereby removing a large amount of the machine-dependent programming and standardizing the computer interface. It is impossible, however to eliminate

all machine dependence because the requirements of the host's operating system must always be satisfied.

As a illustration of this idea, Fig. 4 shows the flow chart of a simple data-channel processor (Ref. 4). It has only a program counter (which points to a list of CAMAC commands), a word count register, and a current address register. A program consists of a random list of commands which conditionally transfers data to or from the memory location specified by the word count register. The processor uses the usual alogorithm of incrementing the word count and current address registers each time a word is transferred. The processor halts when it encounters an EXIT bit microcoded into one of the commands. The host computer is interrupted for buffer servicing whenever a word count overflows. The processor is also designed to make branches and conditional skips depending on the Q-response from the transfer and thereby execute DO loops. More sophisticated processors, such as the Rice-Los Alimos (Ref. 5) processor can be programmed to perform all of the real-time driver operations.

SUMMARY

The full benefits of adopting standardized process interfaces are only realized when the software is also modularized and integrated into modern multitask operating systems. This has been impossible in the past because of the large number of conflicting requirements arising out of various applications and design philosophies, but the software standards groups have now established the types of transactions that need to be supported. These have proven to be fairly simple and, partly because of the use of standardized hardware, easy to implement. The necessary logical processors can be placed in the handler either when it is written or at run time through I/O directives. Alternately, they can be built into a high-performance I/O processor. The problem will be optimally solved, however only when the software requirements have been incorporated uniformly into the hardware.

REFERENCES

1. D. W. Zobrist, et al., "Software Standards and CAMAC .... A Realtime Demonstration," Instru. Tech., vol. 22, pp. 33-38, 1975.

2. R. Setter, "RSX-11D Handler Task for I/O through the CAMAC Serial Highway," Unpublished (1975).

3. Douglas Dowden, "The Design of a General Purpose CAMAC Handler," Unpublished (1975).

4. R. Bercaw, T. Fessler, and J. Arnold, "A Programmable Computer Interface for CAMAC," NASA TN D-7148, Mar. 1973.

5. J. A. Buchanan and H. V. Jones, "CAMAC Multi-Microprogrammed I/O Processor," IEEE Trans. Nuc. Sci., vol. 19, pp. 682-688, 1972.

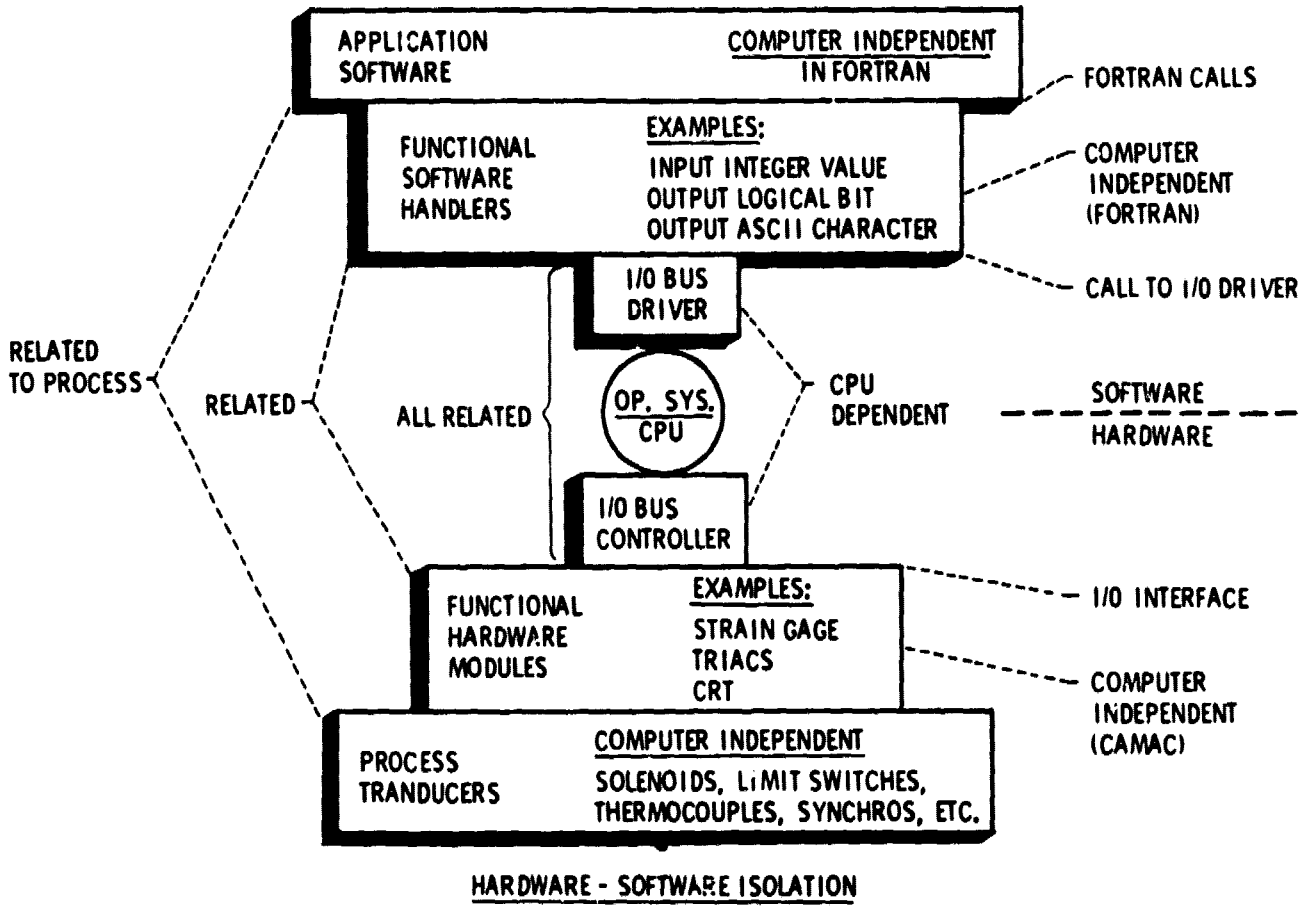

Figure 1. - The Zobrist Sandwich.

E-8702

PRECEDING PAGE BLANK NOT FILMED

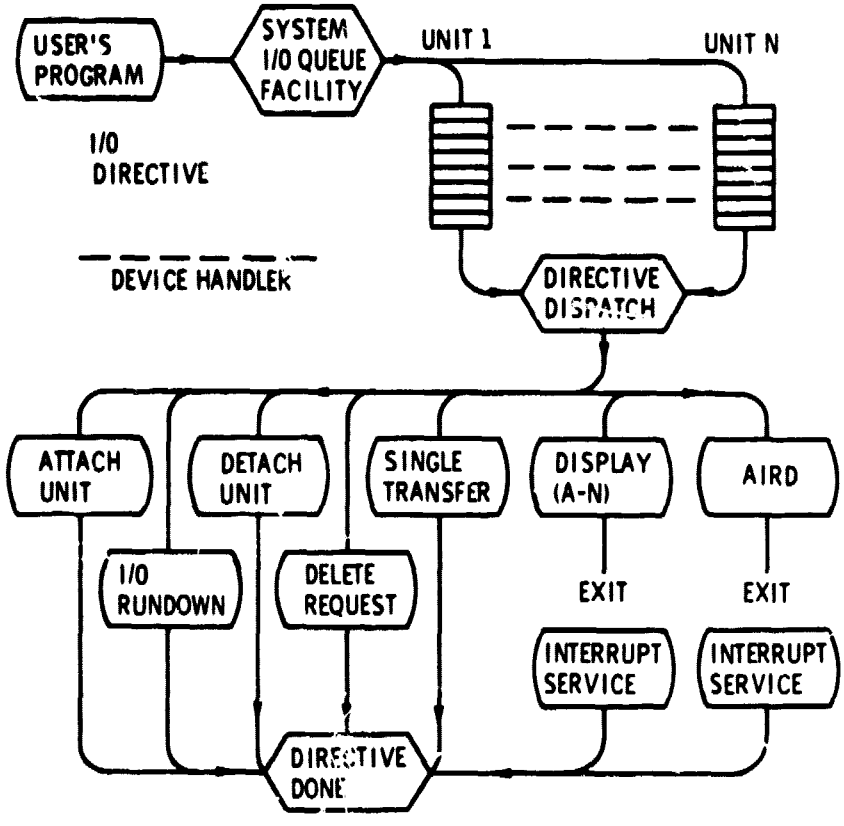

Figure 2. - QUEUED I/O facility.

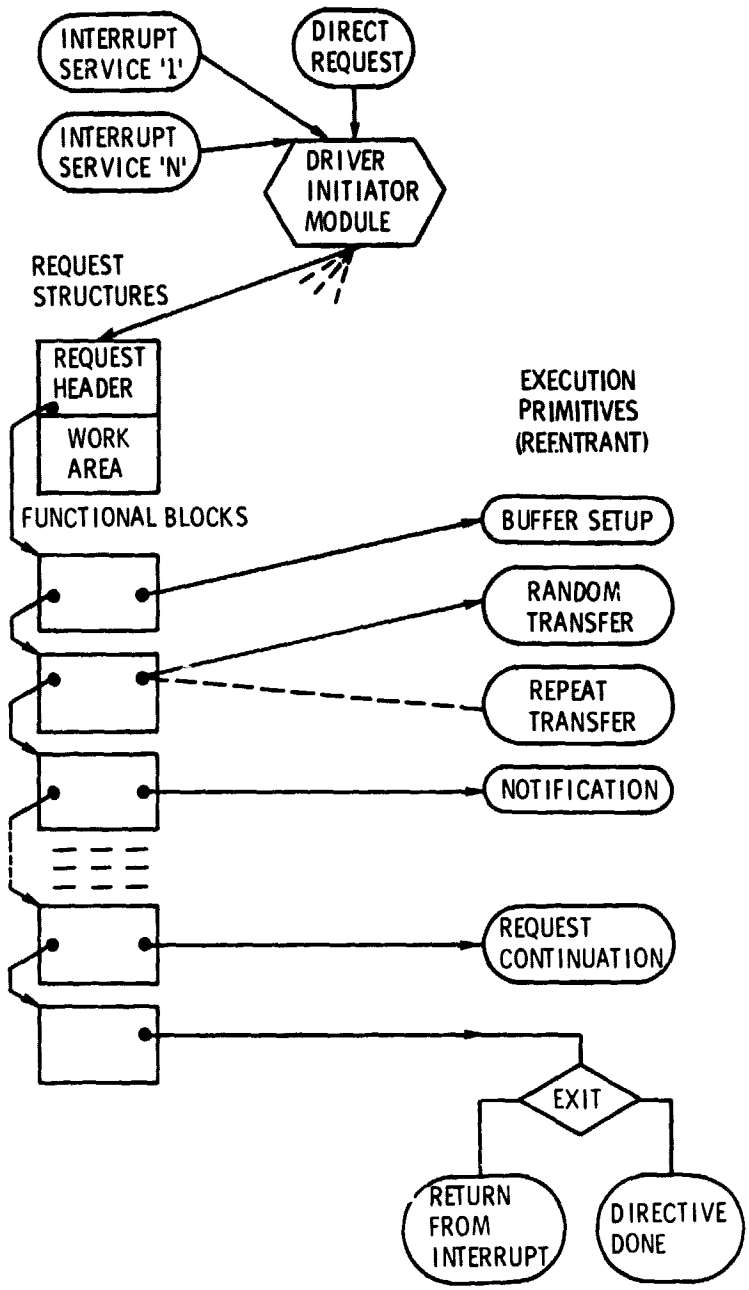

Figure 3. - Modular I/O execution structure.

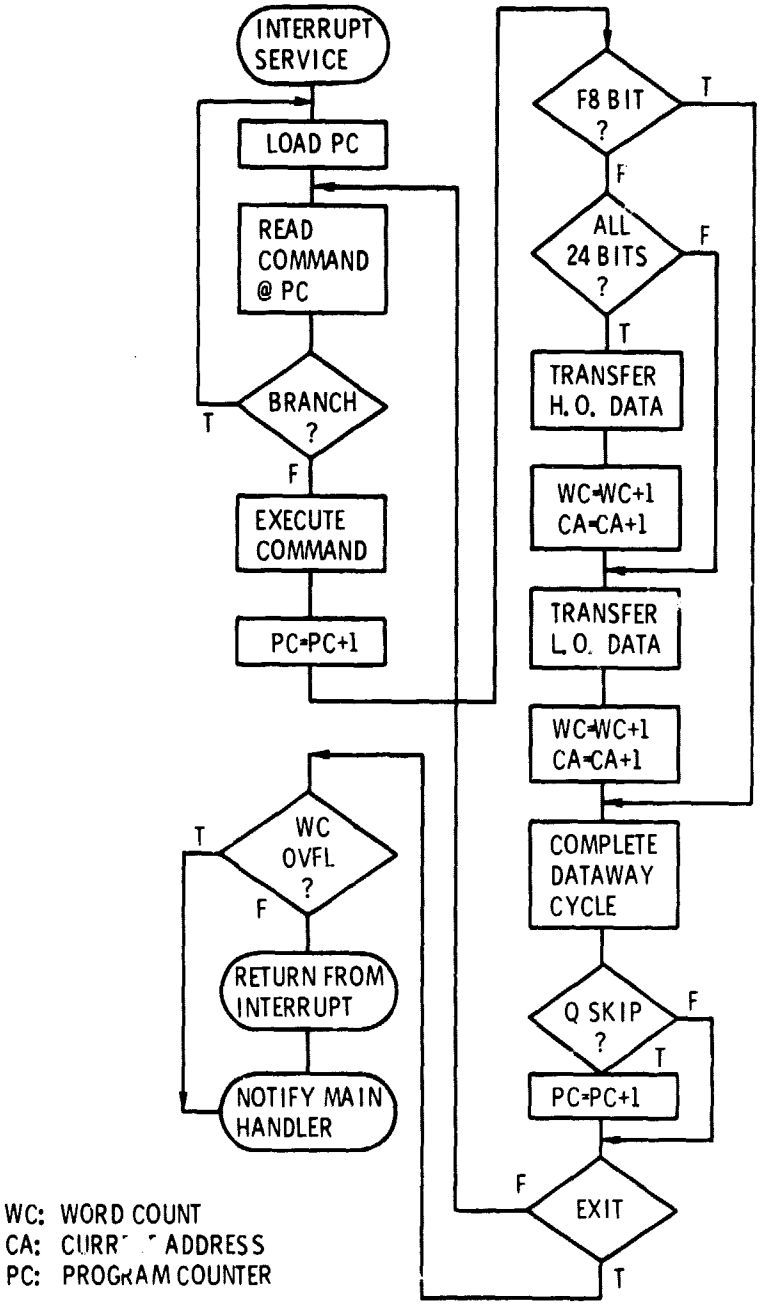

Figure 4. - Data transfer primitive.