Date: November 23, 1971

From (location and U.S. mail address): Poughkeepsie
P. O. Box 390

Dept. & Bldg.: D74/707-1

Telephone Ext.: 3-0051

DEC 03 RECD

IBM

Subject: ADI, EDI, NMI Rationale

Reference:

To: File

I have been asked very frequently of late to explain the purpose of the ADI, EDI and NMI as the major FS interfaces. I offer the attached to document the role which I believe they play and the objectives which I believe they must meet.

If any of you have any comments, questions, or objections to this document, I urge you to see me about them. Please circulate it freely.

George Radin

GR:lmv
Attachment

cc: A. S. Buchman
R. P. Case
C. Conti
L. Cohen
J. Earle
D. J. Gavis
R. Goldberg
O. W. Johnson, Jr.
R. A. Larner
A. Lett
K. MacGuire
A. Magdall
A. Peacock
R. E. Ruthrauff
P. R. Schneider
W. Spruth
E. Sussenguth
R. Talmadge
R. L. Weis

DEC 6 RECD

Why the ADI?

Consider the great many ways people will access the FS system. Non-programmers will push buttons, type simple requests, enter data, etc. Special purpose languages will support more complex accessing requirements (e.g., specialty-oriented computing languages, query languages, data describing languages, operator's languages, report generating languages). General purpose languages will provide applications (and "system") programmers a wide, useful facility for expressing highly complex algorithms, data structures, and control structures.

One way to support these vastly different user interfaces (and the way which is generally used today) is to define a common executing environment at a very low level in the system (e.g., S/370 instructions plus OS control program facilities, or the EDI in FS). Each user interface, is then independently supported on this common law level interface, either by compilation or direction execution of the request. Commonality, communication, etc. between user interfaces happens locally by private agreement between the implementors. One has, then, a query subsystem, an APL subsytem, a COBOL subsystem, each built on the EDI.

Leaving aside, even, the ensuing redundant implementation of common facilities, we are faced with the following unfortunate situation:

Very few applications can be totally implemented by appealing to only one interface as they exist today. Data base applications require the description of data encodings, relationships, authorization, etc., plus the specification of algorithms for processing the data, plus the initiation of cooperating processes for responding to requests, plus the implementation of natural user interfaces for accessing the data. Even simple batch scientific applications require programming in at least one procedural language plus one command language plus "link edit" language.

Thus, a group at an installation attempting to implement an application is faced with two difficult obstacles:

- they must learn several interfaces;
- they must understand the relationships between the interfaces (e.g., between a DD in JCL and a FILE in PL/I, between a CALL in FORTRAN and a CALL in Assembly language, between a FILE in COBOL and a file in IMS).

These relationships are complex, often unnecessarily incompatible, and incompletely defined. The group often finds that the only way to get around this morass is to go beneath it, to the level at which there is a single, full function, performance/RAS predictable interface. Unfortunately, in current systems this interface is difficult and very expensive to program, test, maintain, etc.

A possible solution to this problem is to define one or more new user interfaces which have the characteristic of being adequate and easy to implement complete, stand-alone applications. Thus, we could define a programming language adequate for the total expression of a DB/DC applications, an FMS/FORTRAN interface natural and adequate for FORTRAN kinds of applications, etc. This approach is quite attractive and should not be overlooked.

It has, however, at least two deficiencies. First, we are often faced with existing programs in existing languages which must be incorporated into a total application implementation, we have a large body of programmers who already know existing languages and would be reluctant to unlearn them, and we have commitments to support external standard languages. Second, applications generally do not split easily into disjoint categories. The FORTRAN application wants to go to the data base or produce a report; the data base administrator wants to do a factor analysis on usage statistics, etc.

The ADI is an attempt to overcome these difficulties. It is an internal interface in the same sense as "S/370 instructions plus Control Program Facilities". It is a much higher level hence easier and cheaper to use directly. It is the common base on which all other interfaces are implemented. (These may be implemented by compiling to the ADI, for instance, COBOL, or by programs which run on the ADI and support the interface interpretively, as for instance, a query language.) Thus the ADI forces a high, well-defined level of compatibility between these interfaces.

The ADI clearly must meet at least three objectives:

- It must be enough better than an EDI level in terms of ease of programming to warrant the move up of our most firm, durable interface.

- It must contain enough function, measurability, RAS, etc. to allow applications to be completely implemented on it.

- It must allow applications to be written which can be run with acceptable performance. (This includes old batch programs, large, many-terminal applications, etc.)

Thus the ADI, if successful in meeting these objectives, will achieve our prime goal of allowing new applications to be implemented quickly, easily, predictably, etc. But there are at least two other significant benefits which will come along with this:

- Much redundant implementation effort will be avoided (e.g., optimizing compilers, scientific subroutines, interrupt handling). Also all new function offered at the ADI will be immediately available to all users. This will considerably help the financial justification of such new function.

- We will have significantly more flexibility in making changes below the ADI interface (e.g., hardware/software tradeoffs, new devices, improved algorithms, new technology).

An area of great confusion seems to be in distinguishing the role of the ADI as this internal interface and the role of a concrete syntax of the ADI as a universal programming language. It does, in fact, play both these roles in precisely the same way as S/370 principles of operations is an internal interface and Assembly Language is a universal programming language. The extent to which new applications will be programmed directly in the ADI as opposed to, say, PL/I plus some command language, is hard to predict and not critical at the present time.

The EDI, NMI Relationship

While the prime purpose of an externally visible interface like the ADI is clear in its usefulness to users, it is more difficult to justify the architecture of an internal interface. It can serve at least three purposes:

- to improve the modularity, hence changeability, of the implementation (i.e., changes can be made at one side of the interface without affecting the other side if they do not affect the interface itself).

- to permit multiple implementations independently on each side (where the number of implementations on one side is independent of the number on the other side);

- simply to permit parallel, simultaneous development on either side (i.e., as a management facility to permit decentralized implementation).

The EDI and the NMI serve all these purposes. Let us consider first the EDI. We have concluded early that FS would be characterized by highly integrated physical packaging at both the chip and the "field replaceable unit" level. We concluded also that more than one location would be assigned to building main frames and that these assignments would likely factor by processor MIPS (or revenue) parameters.

Thus, we needed a well-defined interface for the parallel work assignments. If this was our only consideration, we could have defined an I/O attachment architecture and simply instructed each location to build an ADI machine. We observed, however, that there would then be a great deal of duplicated implementations.

In particular, we concluded that it would be highly advantageous to define a logical machine which would be functionally invariant under different processor/memory/storage/I/O configurations but which would be at a much lower level (i.e., easier to build) than the ADI. We could then support the ADI interface by some combination of compilation, interpretations and library. These programs would be usable by all FS systems and impervious to changes in the implementations of the machines below them. We call the interface the EDI. Its purpose, thus, is to define a level which effectively masks all physical configuration differences from programs above it.

Continuing this argument, we could have given the EDI interface to each "main frame" building location and told them to build EDI machines. We observed, however, again much potential duplication of effort. The implementations, for instance, which support the EDI view of source/sink I/O or of storage, are largely independent of the processor MIPS capability and could, therefore, be implemented just once for all processors. We, therefore, defined an interface above which programs could be written which would run correctly on all processors, and called it the NMI.

Consider the motivations for raising or lowering the NMI. If a facility (usable either by the EDI only, or by the ADI programs, or both) can be implemented differently by some processor to good advantage (e.g., performance, cost), then this facility should be placed below the NMI. If on the other hand, advantages can be gained by having the facility implemented in a standard way (e.g., cost, changeability, control), it should be placed above the NMI.

Note that we have chosen to define the major internal interfaces horizontally, or hierarchically. (We could have defined functional subsystems as our major interfaces.) This was motivated by our decision at one end to define the user interface as a single horizontal "machine" interface, and to give total main frame hardware responsibility to the same location at the other end.