11-3Ø-66 K638

TO: TED JOHNSON / NICK MAZZARESE / KEN OLSEN
FROM: GERRY MOORE

WE ARE GETTING THOROUGHLY FRUSTRATED BY LACK OF SUPPORT FROM MAYNARD, PARTICULARLY AS REGARDS DELIVERY AND NOTIFICATION OF DELIVERY CHANGES. ALMOST ANYTHING YOU CAN NAME IS LATE AND EVERYTHING HAS AN EXCUSE (TU55 - NO CPS MOTORS, ASR-33 - NO 5Ø CPS MOTORS, A-D CONVERTER - NO A6Ø1S, COMPUTERS - NO MODULES, MODULES - NO ???, PAPER TAPE - NO PAPER TAGE, ETC. ETC.) THE SITUATION WOULD NOT BE AS BAD IF WE COULD SEE A TREND TOWARD IMPROVEMENT. TROUBLE IS, - WE SEE OLD PROBLEMS COME AND GO, AND COME AGAIN. I'M BEGINNING TO LOSE FAITH THAT WE'LL EVER SOLVE OUR MODULE PROBLEMS. SOMETIMES I SUSPECT THAT OUR WEAKNESS (OR PART OF IT) LIES IN OUR PURCHASING DEPT.

THE LATE DELIVERY PROBLEMS BECOME MOST CRITICAL JUST NOW AS MANY OF OUR CUSTOMERS MUST ACTUALLY RECEIVE GOODS AND PAY BY END OF YEAR OR THEY LOSE THEIR MONEY. THIS IS, FOR EXAMPLE, TRUE OF AN ORDER FOR ONE OF THE DESY DECTAPE SYSTEMS WHICH MUST BE IN THEIR HANDS BY DEC. 1Ø OR THEY SAY THEY WILL HAVE TO CANCEL THE ORDER.

A LISTING OF SOME OF THE ORDERS WERE INTERESTED IN IS AS FOLLOWS:

CUSTOMER	DEC ORDER	DATE ORDERED	ORIG. DEL. DATE	NEW DEL. DATE
2ND INSTITUTE DESY	16273	APR. 25	SEPT. 15	?? (1)
DESY	17498	JUN 15	NOV 15	?? (1)
DESY	179ØØ	JUN 15	AUG 15	?? (2)
DESY	17438	JAN 1	??	?? (2)
DESY	17499	JUN 15	??	?? (1)
AACHEN 3RD PHYSICS	13762	DEC 1, 65	JAN 1Ø	?? (2)
AACHEN 3RD PHYSICS	18148	JUL 27	NOV 1	??
AACHEN 3RD PHYSICS	18147	JUL 27	NOV 1	??
AACHEN 1ST PHYSICS	16214	APR 12	JUL 6	?? (1) (2)
AACHEN STRAHLANTRIEBE	18499	OCT 6	FEB. 67	?? (1)
AEG	16296	MAY 2	OCT 15	?? (1)
AEG	16272	APR 25	OCT 1	DEC 2Ø (3)

BONN	18123	JUL 26	NOV 26	?? (1)(2)
BONN	18413	SEP 16	??	??
NIJMEGEN	18190	AUG 2	SEP 30	?? (2)
MAX-PLANCK	18500	OCT 6	DEC 8	??
BODENFORSCHUNG	18457	SEP 30	DEC 10	?? (3)
U. FREIBURG	?	?	?	?? (1)

(1) THESE ORDERS ALL INCLUDE DECTAPE EQUIPMENT
(2) THESE ORDERS ALL INCLUDE TELETYPES
(3) THESE ARE COMPLETE COMPUTERS

SEVERAL OF THE ABOVE ORDERS HAVE ALREADY BEEN PARTIALED.

WOULD SOMEONE PLEASE TAKE THE TIME TO GIVE ME FIRM DELIVERY DATES ON ALL THE ABOVE ORDERS. WE ARE SIMPLY NOR BEING INFORMED OF CHANGES IN DELIVERY DATES.

THE FIRST ORDER LISTED ABOVE (2ND INSTITUTE DESY) IS OUR MOST CRITICAL DECTAPE ORDER. UNLESS THEY RECEIVE THIS IN-HOUSE BY DECEMBER 10, THEY CANNOT PAY BY DEC. 16, THEY ABSOLUTELY LOSE THEIR MONEY AND THE ORDER WOULD HAVE TO BE CANCELLED. THIS WOULD BE QUITE A SLAP IN THE FACE FOR US AT DESY. WHEREAS SOME INSTITUTES WILL ALLOW US TO INVOICE THEM BEFORE THEY RECEIVE GOODS AND WILL ACTUALLY PAY RATHER THAN LOSE THIS YEARS MONEY, 2ND INSTITUTE WILL NOT. I HAVE TALED WITH GEOFF FINCH ABOUT SWAPPING DELIVERY POSITIONS BETWEEN THE 2ND INSTITUE'S DECTAPE ORDER AND FIRT-BROWN'S DECTAPE ORDER. FOR SOME REASON FIRTH-BROWN IS SCHEDULED WAY AHEAD OF DESY. THE FIRTH-BROWN PURCHASE ORDER OF LAST MARCH REQUESTS ONE YEAR DELIVERY OF THEIR PDP-8 WITH DECTAPE. GEOFF IS AGREEABLE TO SWAPPING DESY AND FIRTH-BROWN PROVIDING WE CAN THEN ASSURE THAT FIRTH-BROWN IS NOT LATE (NOT BEYOND MAR 1). PLEASE CONFIRM THAT YOU CAN MAKE THIS SWAP AND CAN ASSURE GEOFF THAT FIRTH-BROWN WILL NOT BE LATE. ALSO, PLEASE CONFIRM THAT YOU CAN GET THE DECTAPE TO DESY (CONTROL AND 4 TRANSPORTS) BY DEC. 10.

NEXT PROBLEM IS PDP-8 FOR BODENFORSCHUNG. AGAIN THIS IS THIS YEARS MONEY AND THEY MAY NOT BE ABLE TO PAY UNLESS THEY GET THE MACHINE IN DECEMBER AS ORIGINALLY QUOTED. LAST WORD WAS THAT IT IS GONG TO BE A MONTH LATE. HERE AGAIN, PERHAPS WE CAN SWAP THE FIRTH-BROWN PDP-8 FOR THE BODENFORSCHUNG PDP-8. LET ME KNOW WHAT YOU CAN DO.

NEXT PROBLEM IS THE COMPLETE AEG PDP-8 SYSTEM WHICH IS WAY OVERDUE (DEC 16272) AND THIS IS A TYPICAL EXAMPLE OF THE WAY ORDERS ARE HANDLED. THE SCHEDULE ON THIS MACHINE WAS SLIPPED 6 WEEKS BECAUSE OF PROBLEMS WITH THE A601 MODULES THAT GO INTO THE 138E. THEN, WHEN THERE WAS A GENERAL SLIPPAGE OF PDP-8 SCHEDULES DUE TO ADDITIONAL MODULE PROBLEMS, ABSOLUTELY NO ATTENTION WAS PAID TO THE FACT THAT IT HAD ALREADY SLIPPED 6 WEEKS. IT HAD NO PRIORITY OVER MACHINES THAT HADN'T PREVIOUSLY SLIPPED. IT IS NOW EXPECTED TO BE 11 WEEKS LATE.

YOU WILL NOTE THAT ONE OF OUR ORDERS FOR A TELETYPE MACHINE IS ONE YEAR OLD (AACHEN, DEC 13762). HERE AGAIN IS AN EXAMPLE OF THE INABILITY OF ADMINISTRATION TO GIVE ANY SPECIAL PRIORITY TO AN ORDER THAT HAS ALREADY SLIPPED THROUGH ERRORS OR PRODUCTION PROBLEMS. WE SENT A TELEX ORDER FOR THIS WHICH WAS CONFIRMED BY MAIL. THESE WERE LOST BY MAYNARD. LATER ON, WHEN THE ORDER WAS FOUND, IT WAS SCHEDULED AS OF THAT DATE. SINCE THEN, IT HAS SLIPPED ALONG WITH ALL THE OTHER TELETYPES. WHEN WILL WE GET IT?

I AM GENERALLY DISAPPOINTED IN THE CORPORATE SUPPORT WE GET. FIFTY CYCLE BUSINESS SEEMS TO BE AN AFTERTHOUGHT IN MAYNARD. JUST AS WE HAVE HAD PROBLEMS WITH DELIVERY OF 50 CPS MOTORS FOR TELETYPES WE ARE HAVING THE SAME PROBLEMS WITH DECTAPES. I THINK THERE MUST BE A WEAKNESS IN OUR PURCHASING, ALTHOUGH I CONFESS I DON'T KNOW. MAYBE THE PROBLEM LIES IN SOME OTHER AREA.

WHEN I VISIT MAYNARD IN JANUARY I WANT TO TALK OVER WITH YOU THE GENERAL QUESTION OF HOW THE EUROPEAN BUSINESS FITS INTO THE CORPORATION PLANS. MY INTEREST IN BEING HERE DEPENDS ON THE CORPORATIONS INTEREST IN EUROPE.

dec INTEROFFICE MEMORANDUM

DATE November 30, 1966

SUBJECT D664 Procurement

TO
Ken Olsen
Harry Mann
Pete Kaufmann
Dick Best
Stan Olsen
Win Hindle
Nick Mazzarese

FROM Henry Crouse

The enclosed summary is the result of the Purchasing Department (Paul McGaunn) negotiations with suppliers for the procurement of our D664 diodes for the period of January 1967 to June 1968.

Henry Crouse

CONTENTS

I. USAGE - HISTORY

Our current usage rate is approximately 1.5 million per month subject to fluctuation between 800,000 and 1,500,000. The actual usage for the period July 1, 1965 to June 30, 1966 was 12,000,000 units.

II. USAGE - PROJECTED

Anticipating an increase of 30%, our estimated usage for a twelve month period would be 17,500,000 units.

III. BIDDERS

The bidding companies are as follows:

Fairchild Semiconductor, Mt. View, California
International Telephone & Telegraph Company, Lawrence, Mass.
Continental Device Corporation, Hawthorne, California
Sylvania, Hillsboro, New Hampshire
General Electric (not bidding at present), Syracuse, N. Y.

IV. CRITERIA FOR SOURCE SELECTION

We have two suppliers at present, ITT and GE. We feel a need for at least one more vendor for insurance. Secondly, the cost is 0.09 and we set a goal of less than 0.08 minimum. Minneapolis Honeywell negotiated a contract for 40,000,000 units @ 0.077 recently.

V. GOALS

1. Assurance of Supply
2. Lowest cost of acquisition.
3. Stocking arrangement to minimize DEC inventory.

VI. BID SUMMARY

Vendor	1M	3M	5M	10M	15M
ITT			.08	.0675	.06
Fairchild		.0825	.075	.074	
Sylvania	.10	.10	.10	.10	.10
Cont. Device	.08	.07	.06		
GE	Not Bidding at Present				

Continental Device Corporation, Hawthorne, California, is an untried source of supply; however, they offered the lowest price and have a good reputation in the industry.

ITT has performed for us as a steady reliable vendor, providing quantity turn-around to meet our emergency needs.

Sylvania is not actively interested in this diode business; however, they are supplying the DD-1 dice to Tom Stockebrand, Strate Production, @ .03/ea.

GE has 2.5 million units left in present order and will meet competitive pricing in the future. Past performance has been satisfactory.

Fairchild, although their price is higher than the other vendors, we feel a need to add for insurance purposes to the role of active suppliers.

VII. CONCLUSION

1. To contract with Fairchild, ITT, and CDC for 25,000,000 devices to be delivered on an 18-24 month period.

The volume assignment is as follows:

CDC	5,000,000	@ 0.06	20%
Fairchild	5,000,000	@ 0.75	20%
ITT	15,000,000	@ 0.06	60%

2. A stocking agreement whereby each vendor would inventory devices for DEC as follows:

Fairchild	100,000	CDC	50,000
ITT	200,000		

3. The average unit price is $0.063. Previous average price was $0.089. $0.063 will be effective January 1, 1967 - Savings $625,000.00.

dec INTEROFFICE MEMORANDUM

DATE November 29, 1966

SUBJECT

TO Ken Olsen

FROM Ted Johnson

If you decide to pay a visit to Scandinavia in the near future, Arnaud de Vitry had some suggestions of people he would very much like you to meet. All of the people are at the Stockholm Enskilda Bank. You have probably been there before but it is not very far from the Opera House in Stockholm and is the bank associated with the Wallenberg family. Two of the people are directly with the bank - Mr. Pedar Bonde and Mr. Marc Wallenberg Jr. Another man is Sven Malström - he has just been made Technical Director of Enskilda Bank and has an office there but he is Chairman of the company LKB, which is associated with Auto Kemie and Arnaud has been working on LKB to buy a large number of computers from us. He claims it could involve up to fifty PDP-8's. Apparently, all of these people have been primed for a visit from you since Arnaud has had a chance to talk about our company with them in the past. I would encourage you to make such a trip if you think it is at all possible - visiting ASEA and the Enskilda Bank - possibly going with our man Kjell Reistedt to Chalmers University and places like SAAB. You might find my notes on ASEA useful if you were to make such a trip and I did receive a letter from the gentleman that paid us a visit earlier this year.

If you are to travel to Stockholm, Arnaud suggests that you go either mid-December or the period between January 8th and 22nd. Apparently, Swedish businessmen take their holidays on, or about, January 23rd and for a period thereafter.

TJ:mr

dec **INTEROFFICE MEMORANDUM**

DATE November 24th, 1966

SUBJECT

TO Ken Olsen - Maynard

FROM Si Lyle - Toronto

Dear Ken:

I have more or less by default got involved in making my views known of how Digital Equipment of Canada should be organized. I have enclosed my latest outline which is the result of a meeting I had with Ted Johnson last week. The scheme is put forward on the basis that out of a number of such proposals a workable organization can be found that will be capable of capturing the Canadian market at this critical time.

Si

Si

SML:mp
Enclosure

dec INTEROFFICE MEMORANDUM

DATE November 24th, 1966

SUBJECT

TO Ted Johnson - Maynard

FROM Si Lyle - Toronto

Dear Ted:

Further to my initial letter to Stan and our conversation, here are my views on how the Canadian operation should be set up.

Canadian Marketing Organization

The best thing that could ever happen to the Canadian company is for it never to happen. This does not mean that we do not need a Canadian company. It is to stress what we need is not a scaled down version of Maynard but an operation that uses a combination of DEC Maynard and a central Canadian organization. I will deal with how the Canadian situation could be integrated with Maynard first and then discuss the Canadian company.

Apart from the border there is really no reason to have various offices in Canada working out of a central location rather than working within the present regional framework in the United States. In other words, sales offices in Halifax, Montreal and Ottawa should work out of the north-east region, Toronto sales office should work out of the same region as Rochester and Ann Arbor, the Winnipeg office should work out of the same region as Minneapolis/St. Paul and the Vancouver office should work out of the north-west sales region.

In incorporating the Canadian operations in the regional scheme full geographical advantage is taken of the fact that most Canadian offices are dealing with customers who are within a few miles of the border. In some cases these customers are actually closer to a U.S. office than they are to a Canadian office. With a regional setup customers, for example, in Niagara Falls could be handled equally by say Rochester and Toronto. The Canadian office included in the U.S. regional setup would eliminate the present difficulty of offices dealing with customers outside of their country. It would be the job of the regional manager to be sensitive to the customers' requirements and thus to be in a position to make sure that his district offices, regardless of whether they are U.S. or Canadian, can properly handle the customer, regardless of whether he is a Canadian or U.S. customer.

Canada is an extremely long thin country, mainly inhabited about 50 miles high and 4,000 miles wide with about 90% of its population living within 50 miles of the U.S./Canadian border. In other words, by simply extending the U.S. northern regions an additional

(con'd)

DATE November 24th, 1966

SUBJECT

TO Ted Johnson

FROM Si Lyle

50 miles or so all of the Canadian sales offices are automatically included in the U.S. regions. The fact that Canadian cities are so close to the U.S. border has caused many industries to organize their communication, transportation and marketing efforts to run north/south as opposed to east/west. An example of how this regional change could be made into a good deal is the case of Vancouver. It is a lot more sensible and economical for Seattle to look after Vancouver than for any of the offices from eastern Canada to do so. It might not work now as the Seattle sales office has the obstacles of crossing a regional border as well as the hypothetical international border set up by DEC. A salesman working out of Seattle trying to handle a Canadian customer has 2 strikes against him. On the other hand if the area, including the Vancouver office, were included in the same region, then the difficulties would be considerably reduced.

The above regional change does not mean the Canadian company should not exist. In fact there are a large number of unique Canadian problems which can only be handled by a Canadian based operation. Thus effectively what I am proposing is that there must be a central Canadian operation, and the U.S. regions could be extended to include the Canadian offices. The regional tie would be for the day to day problems which are identical whether the office is in Canada or the States. There are a lot of common problems, after all regardless of the location of the office we are selling the same products, basically to the same type of people for basically the same type of usage. Problems involving equipment, delivery, customer training, customer confidence in our equipment reliability, marketing support, sales follow-up, customer follow-up, etc. are all the same whether it be a Canadian office or a U.S. office. A well run region would then strengthen both the Canadian and U.S. operations as that region could take advantage of effectively having an additional office.

Now to the uniquely Canadian problems.

The Canadian operation requires a central organization to deal with those problems that are unique to doing business in Canada. The Canadian company in this context should be considerably more

(con'd)

dec INTEROFFICE MEMORANDUM

DATE November 24th, 1966

SUBJECT

TO Ted Johnson

FROM Si Lyle

autonomous than the present setup. Budgets should be jointly established for all facets of the Canadian operation and then maintained by the Canadian company. Some of the main activities of the Canadian company are as follows:-

(1) Equipment

The Canadian company should actively engage in assuring that all of the equipment meets the local electrical standards and making modifications where required.

The Canadian company should also be part of certain areas of product development - such as logic kits.

(2) Material Handling - Modules

It goes without saying that a central module inventory and a bonded warehouse are absolutely essential. The module volume does not warrant each individual Canadian office becoming involved with the border crossing problems. Also a bonded warehouse is essential since some customers do not pay duty and others do, and it would be uneconomical to have bonded warehouses all across the country.

(3) Invoicing - All Goods

All invoicing should be done from a central point and naturally all invoicing should be done in Canadian funds and include those additions applicable to the customer. Since this again requires a fair amount of administration it is essential that it is a central operation rather than spread out amongst the Canadian offices.

(4) Salary - Administration

Salaried administration for sales and field service should be handled from a central location as well as the distribution of pay cheques.

(con'd)

DATE November 24th, 1966

SUBJECT

TO Ted Johnson

FROM Si Lyle

Salary administration needs to be a science in Canada as salaries vary tremendously from one area to another and the differences are not consistent for various types of vocations. A secretary in Toronto must be paid at a different rate than say the same job in Winnipeg. The same applies to engineers, field service, etc. To cope with this the administration must be handled by a Canadian company.

The problem goes much further, however, in the area of sales engineers. The Canadian employment market is not at all like the U.S. which works to DEC's advantage if it knows how to operate within the Canadian employment market. DEC can attract almost anybody it wants in Canada. This is possible because the computer community is incredibly small in Canada and DEC people are respected members of that community. Thus the present staff attract other members of the community. This whole statement might seem immodest but I can assure the reader that this is the case concerning the present DEC Canadian sales group. From this same computer community come the venturesome ones thus DECAN who can attract and influence the community is itself a victim of it as DEC must remain competitive once a person is on staff.

To understand this environment and to be able to compete within it, it is necessary for salary administration to be in the hands of the Canadian company operating within the guide lines of an overall budget.

(5) Office - Administration

A central operation is required to handle the budgeting of the various sales offices as the costs of all goods and services are different than in the U.S. Also for expediency all expenditures such as travel expenses, etc. must be cleared through a central Canadian company.

(6) Material Handling - Non-modules

In addition to the above responsibilities it would be the requirement of the Canadian central operation to monitor and if necessary maintain optimum border crossing points for the various Canadian sales

(con'd)

offices so that shipments of computers and test systems can be made directly to the customer.

As the Canadian sales volume increases and the number of computers increase it is getting less and less realistic to think that all computer shipments to Canadian customers be made via Carleton Place. Canadian transportation and facilities are obviously not up to the standard of the U.S. and it hardly makes sense to ship from Boston to Ottawa and then to Vancouver, when an nearly all U.S. route of Boston/Seattle/Vancouver is possible, which is probably more efficient but what is more important is that it would be easier for those involved with the shipment in Maynard to understand. This might be a good point at which to state that one of the underlying factors in this organizational approach is that it must be admitted by the Canadian operation that it is more important to do things in such a way that the Maynard personnel readily understand rather than beating them on the head trying to make them understand the Canadian way of doing things. After all it will be a long time before either DEC or Canadian DEC are large enough that they can really be looked upon as two separate operations.

Now back to the point.

The sixth point then deals with the fact that if shipment is made via the United States there will be a number of border crossing points required and at each point DECAN will have to maintain working relationships with a custom broker. Although the individual offices will be involved with maintaining their own custom broker, the central operation will be able to play a very significant role in making sure that any ruling they get at one office is maintained throughout the Canadian offices so that discrepancies amongst the brokers can be avoided. The extra dealing with the custom brokers will not be burdensome and in fact will be much easier to handle and will help avoid delays that are now becoming apparent. At present shipping through a Canadian middleman organization, regardless of its level of efficiency, cannot help but put a delay into the system.

(con'd)

DATE November 24th, 1966

SUBJECT

TO Ted Johnson

FROM Si Lyle

(7) Location

Since the central Canadian office will be dealing with primarily administrative problems and module inventory control, it should be set up in a location where it has optimum communication and transportation with Maynard. There is only one city that falls into this category and that is Montreal. Montreal, as a head office operation for the Canadian company, is an excellent location, not just because of its close proximity to Boston but because ultimately the Montreal area should turn out to be the largest DECAN market. The Montreal area has the largest population in Canada, it is the financial centre of Canada, and it is the head office centre in that most Canadian companies have their head administrative offices in Montreal. It has, probably what is even more significant, a trait which certainly is not apparent in the Toronto area and that is the capability to think for itself. Since our prime competition in computers is IBM we will naturally do better in those areas where IBM is weaker. Since the French-Canadian element of Montreal is less likely to fall in love with brand names, he is not taken in by IBM in the same way as the Anglo-Saxon Toronto area. Due to this DEC will ultimately find Montreal much more receptive than Toronto.

In passing it is worth pointing out that the same is true about the rest of Canada as what I have just said about Montreal. Customers in Western Canada and the extreme eastern part of Canada are much more likely to think for themselves and hence are not brainwashed by IBM. However my main point is where our central office should be and because of communication, transportation, etc. the west and far east is out. That leaves Montreal, Ottawa, or Toronto; and Montreal far exceeds the other two. Also I think the ease of transportation and communication between Montreal and Boston is an important factor. The Montreal operation will somehow get closer to the Maynard operation as it should be, as our size at the present time and for a good long time to come, does not warrant divorse between the two operations.

(con'd)

dec INTEROFFICE MEMORANDUM

DATE November 24th, 1966

SUBJECT

TO Ted Johnson

FROM Si Lyle

In a nutshell let me summarize then the Canadian operation. Let us look at it from the point of view of a district office. At the district office level, for all company activities excluding the actual mechanics of doing business, the district manager will answer to the regional manager and the district office field service personnel will answer to the regional field service manager. There would be a Canadian marketing coordinator who would have the responsibility of maintaining contacts and functions between the district offices and the Canadian central office. The district office will therefore look to Maynard for all marketing support, all deliveries (except modules) and will be responsible for clearing the equipment through customs and getting it to the customer. The district office on the other hand will deal with the Canadian central office for all module orders and all module shipments will be made from the Canadian central office directly to the customer. The district offices will not be responsible for billing but they will be responsible for notifying the central office when non-module orders have been shipped from Maynard so that the central office can bill the customer. The district office will also look to the central Canadian operation for control of budgets, salary administration, travelling expenses, etc. In fact I can even summarize the above further by saying that the ultimate operation would be that the district Canadian office would operate within the U.S. region framework when dealing with customer activities and would answer to the Canadian central office when it is involved with inhouse matters.

GENERAL IMPLEMENTATION

The above scheme, or a modification thereof which does not lose the principle, should be incorporated immediately. This would put an end to some of the functions which are now being built up in the Canadian company and I feel that some of these functions are not really necessary. The least necessary of these is the re-checking of computers

(con'd)

DATE November 24th, 1966

SUBJECT

TO Ted Johnson

FROM Si Lyle

before shipping to customers. The movement of the Canadian central operation from Carleton Place to Montreal should also be planned immediately as now is the time to do it before the Montreal office is formed. The actual use of Montreal rather than Carleton Place as a head office location need not be done immediately but all changes should be in that direction. If production is to be carried on in Canada and is to be taken seriously it would be a good deal better if it were done in Montreal where it would be much closer to Canadian suppliers and to Boston.

The next phase of operation which should also start immediately, although it will obviously take many months to complete, is the establishment of more sales offices in Canada, each of which should start off with one salesman and one field service engineer. Sales offices in Halifax, Winnipeg, Edmonton and Vancouver would very quickly pay for themselves.

The essential key to all of this is that DEC is now starting to capture, as evidenced by the Toronto sales, the confidence of companies involved with process control. Since our market does not have a Foxboro and the like, it leaves it wide open to DEC and since Canadian industries are just awakening to the whole concept of automation and since Canadian industry is infinitely larger than Canadian research, it is a market well worth going into. This market is to be found in cities such as Montreal, Vancouver, Toronto and to a lesser extent Winnipeg and Halifax.

This is not to imply that DEC of Canada turns away from its already founded university and research market commitment but it is to stress the point that if DECAN is ever to really grow it must get into the industrial market. The Canadian industrial market is growing faster than the university market and it is unlikely that we will be able to increase our percentage of the university market, in fact in total dollars spent in the area of research and universities in computers, DEC will get a smaller percentage as the main computer

(con'd)

centres will no doubt remain IBM or CDC and will continue to get much larger. It is nice to be able to say we have 3 computers in the University of Toronto and IBM has 1, but 3 PDP-8's is not an awful lot of money when stacked against one 360/91. The trend in the universities for some time to come will be towards the larger machines. On the other hand the industrial market is a much freer market as they are not pressured by a third party to buy any particular brand of computer and also they are much more ready, willing and able to do business with anybody as long as his product is what he needs. And DEC has just this type of product. To capture the industrial market we have to appear knowledgable, competent, reliable and secure. We will never be able to do all these things if we spread ourselves out 50 miles high and 4,000 miles wide, but if we do it within a framework which is already existing then the total DEC organization will always be readily available to the total Canadian market. I would hate to see DEC of Canada lose out on what can be a tremendous market only because the above approach appears too anti-nationalistic. We are now living in the time of common markets and nationalism should play no part in a good sound marketing organization in Canada and the United States. There is only one thing that Canada really needs and that is a good $10,000. U.S. computer.

Si

Si

cc: Ken Olsen
cc: Denny Doyle

dec INTEROFFICE MEMORANDUM

DATE November 22, 1966

SUBJECT Trade Shows

TO Tim McInerney

FROM R. L. Lane

I just want to add a few points to Ken's memo.

(1) Literature, parts and material should be better marked before it leaves Maynard so it can be placed in the correct section of the booth storage.

(2) A secure cabinet with lockable doors should be built either into a back section or one of the dividers.

(3) We should purchase our own speaker system so not to rely on rental options every show. We will soon pay for it and will get better quality.

(4) Consider a portable lounge which can be set up in the storage area for conferences. It should appear neat and planned.

(5) At large shows consider using the local secretaries to staff an answering service at the booth, make reservations, coordinate transportation, meetings, etc.

(6) After each show a report should be made indicating how many handouts of each were passed out, how many inquiries per product line. This way we can learn how much literature we should take to each show and what products are creating the most interest.

(7) I think we should continue with headers, they really add something to the booth. Also, we might consider the equipment setback through the wall so we will have access to the back of the equipment from the storage area.

(8) When we expect people to fill out bingo cards, let's provide a standing-height writing table on a base which can be relocated to accommodate people flow, with chained pen.

(9) Insist on no more surprises at the trade show such as the Berkeley PDP-8S box.

(10) Not encourage people to hold departmental meetings during the show. It justifies their attending but also competes with booth duty and show activities.

CC: T. Johnson
K. Olsen ✓

dec INTEROFFICE MEMORANDUM

DATE November 22, 1966

SUBJECT

TO ALL OFFICE MANAGERS
M. RUDERMAN
K. OLSEN
H. BURKHARDT

FROM DON BARKER, PALO ALTO

Enclosed is a description of BSL's system. They have sold this system to the Clinical Lab. at the University of California Medical Center and a second order is forthcoming from NIH at Washington.

Each system includes a PDP-8 and DECtapes. Perhaps some of your local hospitals would be interested in it.

Good Luck!

Don

DB/mro

BSL

Pricing Information For BSL LABORATORY DATA PROCESSING SYSTEMS

Bulletin 136

1 November 1966

2229 Fourth Street, Berkeley, California 94710

This Bulletin provides pricing information for the Laboratory Data Processing Systems designed and manufactured by Berkeley Scientific Laboratories. A typical BSL laboratory data processing system is shown in figure 1. These systems are constructed around the Laboratory Data Collector and the LDC Data Input Consoles which were designed for digital data acquisition in experimental and clinical laboratories. In the paragraphs below we will try to give you some indication of the cost of these systems and the manner in which they are designed and installed within laboratories.

There is no standard laboratory data processing system, and we doubt there ever will be one. The reason is that the requirements within the individual laboratories vary from one laboratory to another. Laboratories are using a variety of different analytical instruments, semiautomatic analyzers, manual testing procedures, and a great variety of different clinical report formats and procedures. The Laboratory Data Collector and the LDC Data Input Consoles allow us to build a data collection and data processing system which connects directly to the analytical instrument outputs and provides a convenient interface to the technician for recording and reporting of test data and specimen identification. This much of the system is standardized. We designed the LDC Data Input Consoles such that they include standard options to connect to common analytical instruments such as Coulter Counters, spectrophotometers, flamephotometers, and any other instrument which produces an output reading which can be digitized. The remainder of a laboratory system is dictated by the manner in which this test data must be processed and reported. The data processing hardware and computer programs account for about one-half of the cost of most laboratory systems. This is the portion of our systems which must be tailored to the particular laboratory.

Estimates of the cost of a laboratory data processing system such as we design at BSL can be made by the following schedule:

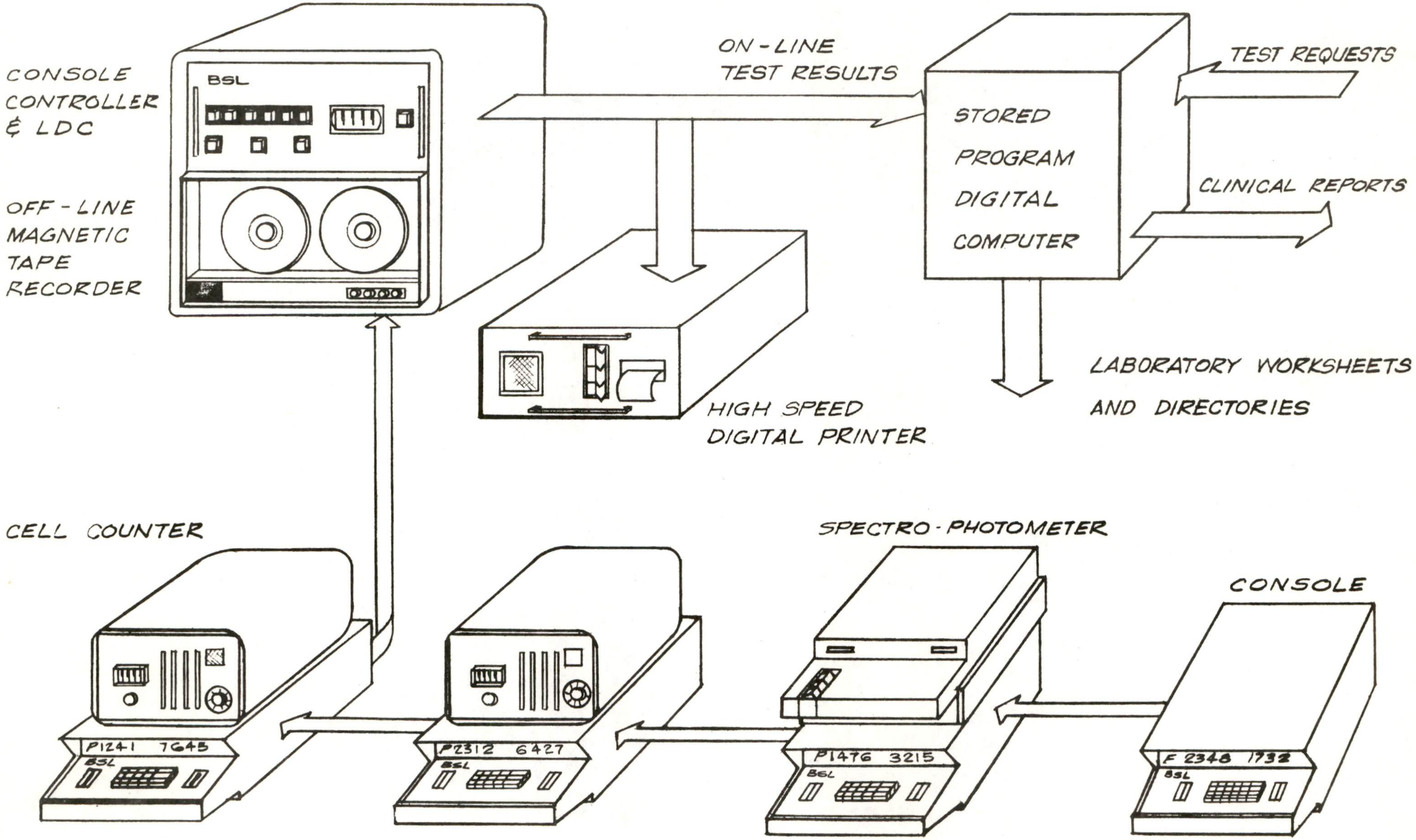


FIG. 1 A BASIC CLINICAL LABORATORY DATA PROCESSING SYSTEM FOR HEMATOLOGY

TEST DATA COLLECTION

We define a test station as one analytical instrument operated by a technician or one semiautomatic instrument. A BSL Data Input Console properly connected to an analytical instrument to form one test station costs approximately $5,500. The Console Control unit which can control up to 20 test stations costs $9,500 and performs all of the operations described in the Laboratory Data Collector brochure attached to this bulletin. A number of Data Input Consoles and the Console Control Unit form a subsystem and provide the means for collecting all of the data from the test stations and presenting it in a form which can be delivered to any digital data recording device and/or a computer. This Test Data Collection subsystem costs $9,500 plus $5,500 per test station. This Test Data Collection subsystem can be made into a complete off-line test data collection and recording system by the addition of a magnetic tape recorder option which costs $7,000. These magnetic tapes are IBM compatible and can be processed on any available computer to produce complete clinical laboratory reports and master files. This is the way in which several laboratories are now starting. They are using available computers several times a day to process the laboratory data recorded on magnetic tapes and produce clinical reports and master files.

DATA PROCESSING

The Clinical Laboratory Systems which we are now delivering include small digital computers which range in cost from $20,000 to $46,000. These computers perform the task of processing the test data and preparing complete clinical laboratory reports on demand or according to pre-assigned schedules. In addition, they perform all the test data calibration and conversion required plus storage of laboratory test information and patient files during the operating day of the clinical laboratory. Finally, they prepare summary tapes and reports for storage of the clinical laboratory reports and transactions which can be used for future reference. We have developed complete programs for a small computer system which costs $46,000 including magnetic tapes and input-output equipment for the generation of clinical reports and input of patient information. Complete clinical laboratory data processing programs are supplied with this computer. The computer and the programs mentioned above are capable of handling test stations in hematology and chemistry.

The typical way in which laboratories are installing the processing systems designed by Berkeley Scientific Laboratories is to begin with a small number of test stations in a particular area of the laboratory, such as hematology, plus the small computer without extensive input-output equipment. This allows them to check out and perfect each of the procedures and train the personnel while they are expanding the system. The basic computer and programs can be supplied for $29,000, and a four station Data Input Console configuration for an additional $31,500. This is a total of $60,500 for a complete computer system which can be expanded up to 20 test stations as well as magnetic tape output and recording for master file processing and storage.

Many recent attempts have been made to use commercial computing equipment and input-output equipment to alleviate the data collection and processing problems within clinical laboratories. However, most of these have failed to make any substantial improvement in the operation of the laboratory because standard commercial computing equipment is by no means designed for collecting test data within a clinical laboratory. Furthermore, the cost of most commercial equipment alone far exceeds the cost of the complete basic system with the computer supplied by Berkeley Scientific Laboratories. We believe that we are the first to design data input consoles specifically for the analytical instruments and the environment found in the laboratory. Secondly, we have written realistic computer programs for performing the laboratory data processing operations rather than attempting to modify or expand elaborate programs produced for entirely different purposes.

We would like to invite you to visit Berkeley Scientific Laboratories and see the equipment which we have designed and the complete systems which we have installed in major clinical and research laboratories. The specifications and the operational requirements of the BSL systems have been specified by scientists and clinicians. They are the best judges as to the appropriateness of any automation system within the clinical laboratory.

dec INTEROFFICE MEMORANDUM

DATE November 21, 1966

SUBJECT Paper Tape Readers

TO K. Olsen✓

FROM J. Smith

We are currently manufacturing two (2) readers - the PR68 for typesetting and the PC01 for general use.

Initial production of the PR68 did run into problems. When released to Production, there was not a sufficient quantity of sprockets on order to meet the expected building rate. Delivery lead time was four (4) months, and the supplier was a single-source manufacturer. I suspect the reason there was not a sufficient quantity of sprockets on order prior to Production release was due to inaccurate forecasting by Sales. This is my opinion only, and it would be difficult to reconstruct situations prior to Production release.

Since Production release, we have managed to reduce the lead time to two (2) months. To date, Purchasing and Engineering have been unsuccessful in finding a second source for sprockets. Current status is as outlined below:

20 - Complete in Stock
35 - Under Construction

The projected usage rate from Sales is sixteen (16) per month. We will meet this rate with little difficulty.

The PC01 was a problem from the very beginning. I must have attended four or five meetings with Engineering and Mike Ford trying to get this unit released to Production. The conclusion at the end of each meeting was that more engineering was required. Last month, reference the attached memo, we agreed to accept building responsibility for the unit. I assigned one of my senior technicians (Dave Ambrose) to investigate the current testing status. He worked on the unit for two or three weeks and through a series of mods and adjustments now has the unit ready for release to Production Testing. If we had not taken the initiative, I am afraid this unit would still be in a state of unacceptance. Current status is as outlined below:

REFERENCE: Attached Memo

1. Section "B": To date this has not been accomplished.

2. The eight (8) remaining units will be tested and delivered by mid December.

3. We have exploded parts for the below requirements submitted by Sales. We feel we can build, test and deliver these quantities.

Jan.	Feb.	March	April	May	June
15	24	30	37	36	38

Sixty cycle units only. No fifty cycle motor has been found by Engineering for use with the punch section.

Jack

JFS/sm

Attachment

dec **INTEROFFICE MEMORANDUM**

DATE October 31, 1966

SUBJECT

TO Nick Mazzarese

FROM Mike Ford

cc: J. Smith
E. Harwood
E. deCastro
B. Dill
J. Jones

As of October 28th, Production accepted responsibility for manufacturing the PC01 high-speed reader/punch with the following provisions:

a. Ed deCastro will be responsible for completing and shipping the first two units. This will be done by the end of next week (November 11).

b. Ed deCastro and Dave Dubay will meet and finalize the acceptance test procedure. This procedure will then be presented to Production. This should be done by November 18th at the latest.

c. Production will commence checking out PC01's in mid-November, and will be responsible for shipping eight units between mid-November and mid-December.

Silence from the above recipients will indicate that these provisions have been recorded correctly.

Mike

ejb

dec INTEROFFICE MEMORANDUM

DATE November 22, 1966

SUBJECT Front Lobby Improvement

TO Ken Olsen

FROM Jim Jordan

The following is a complete list of all items that will improve the lobby short of a complete redesign job.

Asterisks indicate those items which can be achieved immediately at a minimum cost. See the enclosed sketches for visual reference. If you have any questions or comments please call.

STEP I

* Remove all furniture presently in lobby.
* Remove all photos.
* Remove present "DIGITAL" sign over door.
* Lower windowsill at operators area to height of shelf.
* Cover sill with wood grain formica.
Install nonmovable PPG black glass. (Leave 6" open at bottom)
* Provide cabinet for operators to put all personal belongings. (To eliminate clutter)
Install flush bifold doors on closet and in front of PBX
* Clean ceiling tile.
Cover inside and outside of foyer with flush, paintable material.
Refinish both front doors.

STEP II

* Paint all unpaneled walls white.
Install spots (Lightolier, as used in exhibits)
* Hang recent product photos and equipment.
a. Color photos of strate manufacturing.
b. Black & White photo murals on panels.
c. Product (modules, power supplies etc.) mounted on panels similar to above.
* Hang clear plexiglass with logo silk screened-North wall.
Install clock similar to one in Thompson Street lobby - North Wall - $40.00

Gold exhibit carpet - areas in front of North & South Walls.
Six (6) chairs similar to new computer chair. (One to have wheels for receptionist) $66.00 Use LaFonda chairs from exhibits until new ones can be ordered.
* Table from exhibits. (36" round x 18" high)
Provide small phone pedistal.
New L shaped desk. Make tops here. Buy 2 or 3 good looking 2-drawer files.
* Two (2) ash trays (From exhibits)
One (1) trash basket
* New magazine covers.
Install short drapes over desk (West Wall)

dec **INTEROFFICE MEMORANDUM**

DATE November 22, 1966

SUBJECT International Computer Exposition Tim

TO Ken Olsen ✓
Stan Olsen
Nick Mazzarese
Ted Johnson
Mike Ford
Howie Painter
John Jones
Win Hindle

FROM Tim McInerney

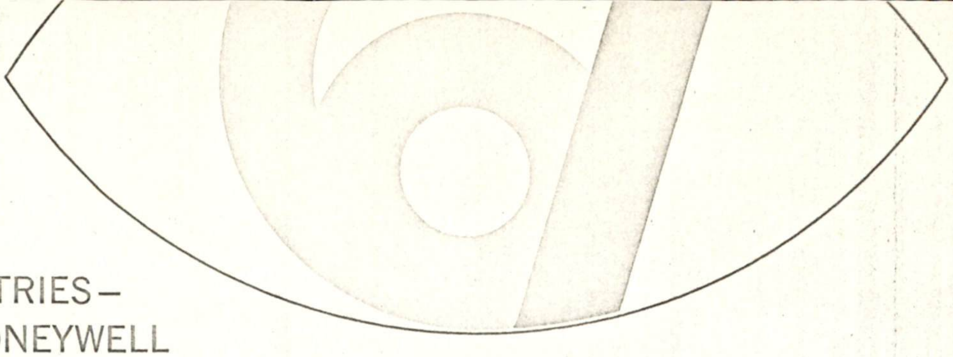
In September I advised you of the subject Exposition, which will take place June 5-8, 1967 at the New York Coliseum. Some of you have requested more information regarding the history of this show and its cost.

Mr. Charles Mathalon of Computer Exposition, Incorporated advised me that since it is not a policy of his Company to circulate lists of participants at this Exposition, he was unable to comply with my request about who would be participating in this Exposition. The only indication that he gave me regarding participants was that the 1967 Exposition is of a highly technical nature and is backed by almost all the major foreign and United States manufacturers in the computer field, who are all participating.

This is the first time for this Exposition, and the expected attendance ranges from 40,000 to 50,000 people. Attached is a copy from a portion of the information brochure forwarded to me by Mr. Mathalon. This shows the expected participants at this Exposition. I hope to obtain a definite list of participants shortly, since Mr. Mathalon is sending one of his representatives here to discuss our participation.

The price of booth space for this Exposition is $5.00 per square foot or $500.00 per 10' booth.

TJM:jdr
Attachment



TRIBUTE TO:

U.S.A.: I.B.M. – G.E. – R.C.A. – LITTON INDUSTRIES – MONROE – UNIVAC – HONEYWELL

JAPAN: HITACHI SEISA KUSHO – MATSUSHITA – MITSUBISHI DENKI – SONY K.K. – TOKYO SHIBAURA DENKI – NEAT ONKYO DENKI

GREAT BRITAIN: E.M.I. ELECTRONICS – ENGLISH ELECTRIC – ELLIOT BROS. – G.E.C. (ELECTRONICS) – MARCONI COMPANY – COMPUTER CONTROLS – PLESSEY COMPANY – SOLARTRON ELECTRONIC GROUP

FRANCE: CIE. FRANCAISE THOMSON-HOUSTON – SOC. D'ELECTRONIQUE ET D'AUTOMATISME – LABORATOIRE CENTRAL DE TELECOMMUNICATIONS – CIE. EUROPEENNE D'AUTOMATISME ELECTRONIQUE – SOS. ALSACIENNE DE CONSTRUCTIONS MECANIQUES – COMPAGNIE I.B.M. FRANCE – COMPAGNIE DES MACHINES BULL

GERMANY: SIEMENS & HALSKE – ZUSE KG ELEKTRONEN-UND RELAIS RECHEN-AUTOMATEN – WEBER ELEKTRONIK – BECKMAN INSTRUMENTS – B. SEIBERT – STANDARD ELEKTRIK LORENZ – BOLKOW APPARATEBAU – DR. ING. PAUL NAMUR KOMMANDITGESELLSCHAFT FUR AUTOMATISIERRUNG

ITALY: ING. C. OLIVETTI – PAOLETTI – O.T.E. SPA – ELETTRONICA S.R.L. – SOC. PER L'ELLETROTECNICA INDUSTRIALE E NAVALE – TECNOMASIO ITALIANA BROWN BOVERI – APPLICAZIONI ELETTRONICE

SWEDEN: STANDARD RADIO & TELEFON – SVENSKA DATAREGISTER – ARENCO AB – ADDO, O FARM V. 12

SWITZERLAND: GUTTINGER AG. FUR ELEKTRONISCHE

HOLLAND: NV. ELECTROLOGICA

THE NEED FOR AN INTERNATIONAL COMPUTER EXPOSITION

The Computer equipment industry is the fastest growing and most dynamic in the country. Exposition '67 gives it an International focus for the presentation of its new developments, products and services . . . to the public and to the buying and planning influences in business, government, education and the highly technical professions.

dec INTEROFFICE MEMORANDUM

DATE 22 November 1966

SUBJECT PDP-9 FORTRAN

TO K. Olsen

FROM Larry Seligman

From the PDP-7 library, PDP-9 inherits a subset (with modifications) of FORTRAN II (basic FORTRAN). We have contracted out a FORTRAN IV compiler to a software house with the contractual stipulation that the compiler meet the ASA standard. A copy of the standards are attached for your convenience.

Most FORTRAN IV compilers for small computers are quite similar since they are written by a small group of software houses using many of the same techniques. What could have set our FORTRAN out above the others would have been the ability to use it effectively with our automatic priority interrupt system.

lhl

(Reprinted with the Permission of Communications of the ACM)

Standards

S. GORN, Editor; R. W. BEMER, Asst. Editor, Glossary & Terminology
E. LOHSE, Asst. Editor, Information Interchange
R. V. SMITH, Asst. Editor, Programming Languages

History and Summary of FORTRAN Standardization Development for the ASA

By W. P. Heising

The American Standards Association (ASA) Sectional Committee X3 for Computers and Information Processing was established in 1960 under the sponsorship of the Business Equipment Manufacturers Association. ASA X 3 in turn established an X3.4 Sectional Subcommittee to work in the area of common programming language standards. On May 17, 1962, X3.4 established by resolution a working group, X3.4.3-FORTRAN to develop American Standard FORTRAN proposals.

RESOLVED:
That X3.4 form a FORTRAN Working Group, to be known as X3.4.3-FORTRAN, with the

Scope. To develop proposed standards of FORTRAN language.

Organization. Shall contain a Policy Committee and a Technical Committee. The Policy Committee will be responsible to X3.4 for the Working Group's mission being accomplished. It will determine general policy, such as language content, and direct the Technical Committee.

Policy Committee Membership. Will be determined by the X3.4 Steering Committee subject to written guidelines which may be amended later and including the following:

a. For each FORTRAN implementation in active development or use, one sponsor voting representative and one user voting representative are authorized.

b. A representative who is inactive may be dropped.

c. Associate members, not entitled to vote but entitled to participate in discussion, are authorized.

Technical Committee. Will develop proposed standards of FORTRAN language under the Policy Committee direction. The Technical Committee will conduct investigations and make reports to the Policy Committee.

On June 25, 1962 invitations to an organizational meeting of X3.4.3 were sent to manufacturers and user groups who might be interested in participating in the development of FORTRAN standards. The first meeting was held August 13–14, 1962 in New York City. X3.4.3 decided to proceed because (1) FORTRAN standarization was needed, and (2) a sufficiently wide representation of interested persons was participating.

A resolution on objectives was adopted unanimously on August 14, 1962.

The objective of the X3.4.3 Working Group of ASA is to produce a document or documents which will define the ASA Standard or Standards for the FORTRAN language. The resulting standard language will be clearly and recognizably related to that language, with its variations, which has been called FORTRAN in the past. The criteria used to consider and evaluate various language elements will include (not in order of importance):

a. Ease of use by humans.
b. Compatibility with past FORTRAN use,
c. Scope of application,
d. Potential for extension,
e. Facility of implementation, i.e. compilation and execution efficiency.

THE FORTRAN standard will facilitate machine-to-machine transfer of programs written in ASA Standard FORTRAN. The Standard will serve as a reference document both for users who wish to achieve this objective and for manufacturers whose programming products will make it possible. The content and method of presentation of the standard will recognize this purpose.”

It was the consensus of the group that (1) there was definite interest in developing a standard corresponding to what is popularly known as FORTRAN IV, and (2) there was interest in developing for small and intermediate computers a FORTRAN standard near the power of FORTRAN II, however suitably modified to be compatible with the associated FORTRAN IV. Accordingly, two Technical Committees, designated X3.4.3-IV and X3.4.3-II respectively, were established to create drafts. Most of the detailed work in developing drafts has been done by technical committees.

The X3.4.3-II Technical Committee completed and approved a draft in May, 1963. A Technical Fact Finding Committee was appointed and reported in August, 1964 on a comparison of the X3.4.3-II approved draft and an approved working draft of the X3.4.3-IV Technical Committee. This brought to light stylistic, terminological, and content differences and conflicts. In April, 1964 the X3.4.3-IV Technical Committee completed a draft of FORTRAN. In June, 1964 X3.4.3 received and compared the two drafts and (1) resolved conflicts in content, and (2) resolved the conflicting style and terminology. This was accomplished by recasting the X3.4.3-II document to reflect the style of the X3.4.3-IV document while retaining the original content. To reduce confusion, X3.4.3 decided to call the languages Basic FORTRAN and FORTRAN.

Editor's Note

The following working documents have been produced by a Subcommittee of the American Standards Association Sectional Committee X3, Computers and Information Processing, in its efforts to develop a proposed American Standard. In order that the final version of the proposed American Standard reflect the largest public consensus, X3 has authorized publication of these documents to elicit comment, criticism and general public reaction with the understanding that such working documents are intermediate results in the standardization process and are subject to change, modification or withdrawal in part or in whole. Correspondence about the documents should be addressed to the X3 Secretary, BEMA, 235 East 42nd Street, New York, N. Y. 10017.—R.V.S.

(Reprinted with Permission of Communications of the ACM)

A Programming Language for Information Processing on Automatic Data Processing Systems

CONTENTS

FORTRAN *vs.* Basic FORTRAN

1. INTRODUCTION

1.1 PURPOSE. This specification establishes the form for and the interpretation of programs expressed in the FORTRAN language for the purpose of promoting a high degree of interchangeability of such programs for use on a variety of automatic data processing systems. A processor shall conform to this specification provided it accepts, and interprets as specified, at least those forms and relationships described herein.

Insofar as the interpretation of the form and relationships described are not affected, any statement of requirement could be replaced by a statement expressing that the specification does not provide an interpretation unless the requirement is met. Further, any statement of prohibition could be replaced by a statement expressing that the specification does not provide an interpretation when the prohibition is violated.

1.2 SCOPE. This specification establishes:

(1) The form of a program written in the FORTRAN language.

(2) The form of writing input data to be processed by such a program operating on automatic data processing systems.

(3) Rules for interpreting the meaning of such a program.

(4) The form of the output data resulting from the use of such a program on automatic data processing systems, provided that the rules of interpretation establish an interpretation.

This specification does not prescribe:

(1) The mechanism by which programs are transformed for use on a data processing system (the combination of this mechanism and data processing system is called a processor).

(2) The method of transcription of such programs or their input or output data to or from a data processing medium.

(3) The manual operations required for set-up and control of the use of such programs on data processing equipment.

(4) The results when the rules for interpretation fail to establish an interpretation of such a program.

(5) The size or complexity of a program that will exceed the capacity of any specific data processing system or the capability of a particular processor.

(6) The range or precision of numerical quantities.

2. BASIC TERMINOLOGY

This section introduces some basic terminology and some concepts. A rigorous treatment of these is given in later sections. Certain assumptions concerning the meaning of grammatical forms and particular words are presented.

A program that can be used as a self-contained computing procedure is called an *executable program* (9.1.6).

1. INTRODUCTION

1.1 PURPOSE. This specification establishes the form for and the interpretation of programs expressed in the FORTRAN language for the purpose of promoting a high degree of interchangeability of such programs for use on a variety of automatic data processing systems. A processor shall conform to this specification provided it accepts, and interprets as specified, at least those forms and relationships described herein.

Insofar as the interpretation of the form and relationships described are not affected, any statement of requirement could be replaced by a statement expressing that the specification does not provide an interpretation unless the requirement is met. Further, any statement of prohibition could be replaced by a statement expressing that the specification does not provide an interpretation when the prohibition is violated.

1.2 SCOPE. This specification establishes:

(1) The form of a program written in the FORTRAN language.

(2) The form of writing input data to be processed by such a program operating on automatic data processing systems.

(3) Rules for interpreting the meaning of such a program.

(4) The form of the output data resulting from the use of such a program on automatic data processing systems, provided that the rules of interpretation establish an interpretation.

This specification does not prescribe:

(1) The mechanism by which programs are transformed for use on a data processing system (the combination of this mechanism and data processing system is called a processor).

(2) The method of transcription of such programs or their input or output data to or from a data processing medium.

(3) The manual operations required for set-up and control of the use of such programs on data processing equipment.

(4) The results when the rules for interpretation fail to establish an interpretation of such a program.

(5) The size or complexity of a program that will exceed the capacity of any specific data processing system or the capability of a particular processor.

(6) The range or precision of numerical quantities.

2. BASIC TERMINOLOGY

This section introduces some basic terminology and some concepts. A rigorous treatment of these is given in later sections. Certain conventions concerning the meaning of grammatical forms and particular words are presented.

A program that can be used as a self-contained computing procedure is called an *executable program* (9.1.6).

An executable program consists of precisely one main program and possibly one or more subprograms (9.1.6).

A *main program* is a set of statements and comments not containing a FUNCTION, SUBROUTINE, or BLOCK DATA statement (9.1.5).

A *subprogram* is similar to a main program but is headed by a BLOCK DATA, FUNCTION, or SUBROUTINE statement. A subprogram headed by a BLOCK DATA statement is called a specification subprogram. A subprogram headed by a FUNCTION or SUBROUTINE statement is called a procedure subprogram (9.1.3, 9.1.4).

The term *program unit* will refer to either a main program or subprogram (9.1.7).

Any program unit except a specification subprogram may reference an *external procedure* (Section 9).

An external procedure that is defined by FORTRAN statements is called a *procedure subprogram*. External procedures also may be defined by other means. An external procedure may be an external function or an external subroutine. An external function defined by FORTRAN statements headed by a FUNCTION statement is called a *function subprogram*. An external subroutine defined by FORTRAN statements headed by a SUBROUTINE statement is called a *subroutine subprogram* (Sections 8 and 9).

Any program unit consists of *statements* and *comments*. A statement is divided into physical sections called *lines*, the first of which is called an *initial line* and the rest of which are called *continuation lines* (3.2).

There is a type of line called a comment that is not a statement and merely provides information for documentary purposes (3.2).

The statements in FORTRAN fall into two broad classes—executable and nonexecutable. The executable statements specify the action of the program while the nonexecutable statements describe the use of the program, the characteristics of the operands, editing information, statement functions, or data arrangement (7.1, 7.2).

The syntactic elements of a statement are *names* and *operators*. Names are used to reference objects such as data or procedures. Operators, including the imperative verbs, specify action upon named objects.

One class of name, the *array name*, deserves special mention. The name and the dimensions of the array of values denoted by the array name are declared prior to use. An array name may be used to identify an entire array. An array name qualified by a subscript may be used to identify a particular element of the array (5.1.3).

Data names and the arithmetic (or logical) operators may be connected into arithmetic (or logical) expressions that develop values. These values are derived by performing the specified operations on the named data (Section 6).

The identifiers used in FORTRAN are names and numbers. Data are named. Procedures are named. Statements are labeled with numbers. Input/output units are numbered or identified by a name whose value is the numerical unit designation (Sections 3, 6, 7).

At various places in this document there are statements with associated lists of entries. In all such cases the list is assumed to contain at least one entry unless an explicit exception is stated. As an example, in the statement

SUBROUTINE $s(a_1, a_2, \cdots a_n)$

it is assumed that at least one symbolic name is included in the list within parentheses. A *list* is a set of identifiable elements each of which is separated from its successor by a comma. Further, in a sentence a plural form of a noun will be assumed to also specify the singular form of that noun as a special case when the context of the sentence does not prohibit this interpretation.

The term *reference* is used as a verb with special meaning as defined in Section 5.

An executable program consists of precisely one main program and possibly one or more subprograms (9.1.6).

A *main program* is a set of statements and comments not containing a FUNCTION or SUBROUTINE statement (9.1.5).

A *procedure subprogram* is similar to a main program but is headed by a FUNCTION or SUBROUTINE statement. A procedure subprogram is sometimes referred to as a subprogram (9.1.3).

The term *program unit* will refer to either a main program or subprogram (9.1.7).

Any program unit may reference an *external procedure* (Section 9).

An external procedure that is defined by FORTRAN statements is called a *procedure subprogram*. External procedures also may be defined by other means. An external procedure may be an external function or an external subroutine. An external function defined by FORTRAN statements headed by a FUNCTION statement is called a *function subprogram*. An external subroutine defined by FORTRAN statements headed by a SUBROUTINE statement is called a *subroutine subprogram* (Sections 8 and 9).

Any program unit consists of *statements* and *comments*. A statement is divided into physical sections called *lines*, the first of which is called an *initial line* and the rest of which are called *continuation lines* (3.2).

There is a type of line called a comment that is not a statement and merely provides information for documentary purposes (3.2).

The statements in FORTRAN fall into two broad classes—executable and nonexecutable. The executable statements specify the action of the program while the nonexecutable statements describe the use of the program, the characteristics of the operands, editing information, statement functions, or data arrangement (7.1, 7.2).

The syntactic elements of a statement are *names* and *operators*. Names are used to reference objects such as data or procedures. Operators, including the imperative verbs, specify action upon named objects.

One class of name, the *array name*, deserves special mention. The name and the dimensions of the array of values denoted by the array name are declared prior to use. An array name may be used to identify an entire array. An array name qualified by a subscript may be used to identify a particular element of the array (5.1.3).

Data names and the arithmetic operators may be connected into arithmetic expressions that develop values. These values are derived by performing the specified operations on the named data (Section 6).

The identifiers used in FORTRAN are names and numbers. Data are named. Procedures are named. Statements are labeled with numbers. Input/output units are numbered or identified by a name whose value is the numerical unit designation.

At various places in this document there are statements with associated lists of entries. In all such cases the list is assumed to contain at least one entry unless an explicit exception is stated. As an example, in the statement

SUBROUTINE $s(a_1, a_2, \cdots, a_n)$

it is assumed that at least one symbolic name is included in the list within parentheses. A *list* is a set of identifiable elements, each of which is separated from its successor by a comma. Further, in a sentence a plural form of a noun will be assumed to also specify the singular form of that noun as a special case when the context of the sentence does not prohibit this interpretation.

The term *reference* is used as a verb with special meaning as defined in Section 5.

3. PROGRAM FORM

Every program unit is constructed of characters grouped into lines and statements.

3.1 The Fortran Character Set. A program unit is written using the following characters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and:

Character	*Name of Character*
	Blank
=	Equals
+	Plus
−	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point
$	Currency Symbol

The order in which the characters are listed does not imply a collating sequence.

3.1.1 *Digits.* A digit is one of the ten characters: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Unless specified otherwise, a string of digits will be interpreted in the decimal base number system when a number system base interpretation is appropriate.

An octal digit is one of the eight characters: 0, 1, 2, 3, 4, 5, 6, 7. These are only used in the STOP (7.1.2.7.1) and PAUSE (7.1.2.7.2) statements.

3.1.2 *Letters.* A letter is one of the twenty-six characters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z.

3.1.3 *Alphanumeric Characters.* An alphanumeric character is a letter or a digit.

3.1.4 *Special Characters.* A special character is one of the eleven characters blank, equals, plus, minus, asterisk, slash, left parenthesis, right parenthesis, comma, decimal point, and currency symbol.

3.1.4.1 *Blank Character.* With the exception of the uses specified (3.2.2, 3.2.3, 3.2.4, 4.2.6, 5.1.1.6, 7.2.3.6, and 7.2.3.8), a blank character has no meaning and may be used freely to improve the appearance of the program subject to the restriction on continuation lines in 3.3.

3.2 Lines. A line is a string of 72 characters. All characters must be from the Fortran character set except as described in 5.1.1.6 and 7.2.3.8.

The character positions in a line are called columns and are consecutively numbered 1, 2, 3, ··· , 72. The number indicates the sequential position of a character in the line starting at the left and proceeding to the right.

3.2.1 *Comment Line.* The letter C in column 1 of a line designates that line as a comment line. A comment line must be immediately followed by an initial line, another comment line, or an end line.

A comment line does not affect the program in any way and is available as a convenience for the programmer.

3.2.2 *End Line.* An end line is a line with the character blank in columns 1 through 6, the characters E, N, and D, once each and in that order, in columns 7 through 72, preceded by, interspersed with, or followed by the character blank. The end line indicates to the processor, the end of the written description of a program unit (9.1.7). Every program unit must physically terminate with an end line.

3.2.3 *Initial Line.* An initial line is a line that is neither a comment line nor an end line and that contains the digit 0 or the character blank in column 6. Columns 1 through 5 contain the statement label or each contains the character blank.

3. PROGRAM FORM

Every program unit is constructed of characters grouped into lines and statements.

3.1 The Fortran Character Set. A program unit is written using the following characters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and:

Character	*Name of Character*
	Blank
=	Equals
+	Plus
−	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point

The order in which the characters are listed does not imply a collating sequence.

3.1.1 *Digits.* A digit is one of the ten characters: 0, 1. 2, 3, 4, 5, 6, 7, 8, 9. Unless specified otherwise, a string of digits will be interpreted in the decimal base number system when a number system base interpretation is appropriate.

An octal digit is one of the eight characters: 0, 1, 2, 3, 4, 5, 6, 7. These are only used in the STOP (7.1.2.7.1) and PAUSE (7.1.2.7.2) statements.

3.1.2 *Letters.* A letter is one of the twenty-six characters: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z.

3.1.3 *Alphanumeric Characters.* An alphanumeric character is a letter or a digit.

3.1.4 *Special Characters.* A special character is one of the ten characters: blank, equals, plus, minus, asterisk, slash, left parenthesis, right parenthesis, comma, and decimal point.

3.1.4.1 *Blank Character.* With the exception of the uses specified (3.2.2, 3.2.3, 3.2.4, 7.2.3.6, and 7.2.3.8), a blank character has no meaning and may be used freely to improve the appearance of the program subject to the restriction on continuation lines in 3.3.

3.2 Lines. A line is a string of 72 characters. All characters must be from the Fortran character set except as described in 7.2.3.8.

The character positions in a line are called columns and are consecutively numbered 1, 2, 3, ··· , 72. The number indicates the sequential position of a character in the line starting at the left and proceeding to the right.

3.2.1 *Comment Line.* The line C in column 1 of a line designates that line as a comment line. A comment line must be immediately followed by an initial line, another comment line, or an end line.

A comment line does not affect the program in any way and is available as a convenience for the programmer.

3.2.2 *End Line.* An end line is a line with the character blank in columns 1 through 6, the characters E, N, and D, once each in that order, in columns 7 through 72, preceded by, interspersed with, or followed by the character blank. The end line indicates, to the processor, the end of the written description of a program unit (9.1.7). Every program unit must physically terminate with an end line.

3.2.3 *Initial Line.* An initial line is a line that is neither a comment line nor an end line and that contains the digit 0 or the character blank in column 6. Columns 1 through 5 contain the statement label or each contains the character blank.

3.2.4 *Continuation Line*. A continuation line is a line that contains any character other than the digit 0 or the character blank in column 6, and does not contain the character C in column 1.

A continuation line may only follow an initial line or another continuation line.

3.3 Statements. A statement consists of an initial line optionally followed by up to nineteen ordered continuation lines. The statement is written in columns 7 through 72 of the lines. The order of the characters in the statement is columns 7 through 72 of the initial line followed, as applicable, by columns 7 through 72 of the first continuation line, columns 7 through 72 of the next continuation line, etc.

3.4 Statement Label. Optionally, a statement may be labeled so that it may be referred to in other statements. A statement label consists of from one to five digits. The value of the integer represented is not significant but must be greater than zero. The statement label may be placed anywhere in columns 1 through 5 of the initial line of the statement. The same statement label may not be given to more than one statement in a program unit. Leading zeros are not significant in differentiating statement labels.

3.5 Symbolic Names. A symbolic name consists of from one to six alphanumeric characters, the first of which must be alphabetic. See 10.1 through 10.1.10 for a discussion of classification of symbolic names and restrictions on their use.

3.6 Ordering of Characters. An ordering of characters is assumed within a program unit. Thus, any meaningful collection of characters that constitutes names, lines, and statements exists as a totally ordered set. This ordering is imposed by the character position rule of 3.2 (which orders characters within lines) and the order in which lines are presented for processing.

4. DATA TYPES

Six different types of data are defined. These are integer, real, double precision, complex, logical, and Hollerith. Each type has a different mathematical significance and may have different internal representation. Thus the data type has a significance in the interpretation of the associated operations with which a datum is involved. The data type of a function defines the type of the datum it supplies to the expression in which it appears.

4.1 Data Type Association. The name employed to identify a datum or function carries the data type association. The form of the string representing a constant defines both the value and the data type.

A symbolic name representing a function, variable, or array must have only a single data type association for each program unit. Once associated with a particular data type, a specific name implies that type for any differing usage of that symbolic name that requires a data type association throughout the program unit in which it is defined.

Data type may be established for a symbolic name by declaration in a type-statement (7.2.1.6) for the integer, real, double precision, complex, and logical types. This specific declaration overrides the implied association available for integer and real (5.3).

There exists no mechanism to associate a symbolic name with the Hollerith data type. Thus data of this type, other than constants, are identified under the guise of a name of one of the other types.

4.2 Data Type Properties. The mathematical and the representation properties for each of the data types are defined in the following sections. For real, double precision, and integer data, the value zero is considered neither positive nor negative.

4.2.1 *Integer Type*. An integer datum is always an exact representation of an integer value. It may assume positive, negative, and zero values. It may only assume integral values.

3.2.4 *Continuation Line*. A continuation line is a line that contains any character other than the digit 0 or the character blank in column 6, and does not contain the character C in column 1.

A continuation line may only follow an initial line or another continuation line.

3.3 Statements. A statement consists of an initial line optionally followed by up to five ordered continuation lines. The statement is written in columns 7 through 72 of the lines. The order of the characters in the statement is columns 7 through 72 of the initial line followed, as applicable, by columns 7 through 72 of the first continuation line, columns 7 through 72 of the next continuation line, etc.

3.4 Statement Label. Optionally, a statement may be labeled so that it may be referred to in other statements. A statement label consists of from one to four digits. The value of the integer represented is not significant but must be greater than zero. The statement label may be placed anywhere in columns 1 through 5 of the initial line of the statement. The same statement label may not be given to more than one statement in a program unit. Leading zeros are not significant in diffentiating statement labels.

3.5 Symbolic Names. A symbolic name consists of from one to five alphanumeric characters, the first of which must be alphabetic. See 10.1 through 10.1.10 for a discussion of classification of symbolic names and restrictions on their use.

3.6 Ordering of Characters. An ordering of characters is assumed within a program unit. Thus, any meaningful collection of characters that constitutes names, lines, and statements exists as a totally ordered set. This ordering is imposed by the character position rule of 3.2 (which orders characters within a line) and the order in which lines are presented for processing.

4. DATA TYPES

Two different types of data are defined. These are integer and real. Each type has a different mathematical significance and may have different internal representation. Thus the data type has a significance in the interpretation of the associated operations with which a datum is involved. The data type of a function defines the type of the datum it supplies to the expression in which it appears.

4.1 Data Type Association. The name employed to identify a datum or function carries the data type association. The form of the string representing a constant defines both the value and the data type.

A symbolic name representing a function, variable, or array must have only a single data type association for each program unit. Once associated with a particular data type, a specific name implies that type for any differing usage of that symbolic name that requires a data type association throughout the program unit in which it is defined.

Data type is established for a symbolic name by the first character of that name (5.3).

4.2 Data Type Properties. The mathematical and the representation properties for each of the data types are defined in the following sections. For both real and integer data, the value zero is considered neither positive nor negative.

4.2.1 *Integer Type*. An integer datum is always an exact representation of an integer value. It may assume positive, negative, and zero values. It may only assume integral values.

4.2.2 *Real Type.* A real datum is a processor approximation to the value of a real number. It may assume positive, negative, and zero values.

4.2.3 *Double Precision Type.* A double precision datum is a processor approximation to the value of a real number. It may assume positive, negative, and zero values. The degree of approximation, though undefined, must be greater than that of type real.

4.2.4 *Complex Type.* A complex datum is a processor approximation to the value of a complex number. The representation of the approximation is in the form of an ordered pair of real data. The first of the pair represents the real part and the second, the imaginary part Each part has, accordingly, the same degree of approximation as for a real datum.

4.2.5 *Logical Type.* A logical datum may assume only the truth values of true or false.

4.2.6 *Hollerith Type.* A Hollerith datum carries symbolic information (as opposed to a numeric or logical value). The symbolic information may consist of any symbol combination capable of representation in the processor. The representation for blank is a valid and significant character in a Hollerith datum.

5. DATA AND PROCEDURE IDENTIFICATION

Names are employed to reference or otherwise identify data and procedures.

The term *reference* is used to indicate an identification of a datum implying that the current value of the datum will be made available during the execution of the statement containing the reference. If the datum is identified but not necessarily made available, the datum is said to be *named*. One case of special interest in which the datum is named is that of assigning a value to a datum, thus defining or redefining the datum.

The term, reference, is used to indicate an identification of a procedure implying that the actions specified by the procedure will be made available.

A complete and rigorous discussion of reference and definition, including redefinition, is contained in Section 10.

5.1 Data and Procedure Names. A data name identifies a constant, a variable, an array or array element, or a block (7.2.1.3). A procedure name identifies a function or a subroutine.

5.1.1 *Constants.* A constant is a name that references a value or symbolic information derived from the name. A constant may not be redefined.

An integer, real, or double precision constant is said to be signed when it is immediately preceded by a plus or minus. Also, for these types, an optionally signed constant is either a constant or a signed constant.

5.1.1.1 *Integer Constant.* An integer constant is formed by a nonempty string of digits. The datum formed this way is interpreted as the value represented by the digit string.

5.1.1.2 *Real Constant.* A basic real constant is formed by an integer part, a decimal point, and a decimal fraction part in that order. Both the integer part and the decimal fraction part are formed by a string of digits; either one of these strings may be empty, but not both. The datum formed this way is interpreted as representing a value that is an approximation to the number represented by the integer and fraction parts.

A decimal exponent is formed by the letter E followed by an optionally signed integer constant. This exponent is interpreted as a multiplier (to be applied to the constant immediately preceding it) that is an approximation to ten raised to the power specified by the field following the E .

A real constant is a basic real constant, a basic real constant followed by a decimal exponent, or an integer constant followed by a decimal exponent.

5.1.1.3 *Double Precision Constant.* A double precision exponent is formed and interpreted identically to a decimal exponent except that the letter D is used instead of the letter E .

4.2.2 *Real Type.* A real datum is a processor approximation to the value of a real number. It may assume positive, negative, and zero values.

4.2.3

4.2.4

4.2.5

4.2.6

5. DATA AND PROCEDURE IDENTIFICATION

Names are employed to reference or otherwise identify data and procedures.

The term *reference* is used to indicate an identification of a datum implying that the current value of the datum will be made available during the execution of the statement containing the reference. If the datum is identified but not necessarily made available, the datum is said to be *named*. One case of special interest in which the datum is named is that of assigning a value to a datum, thus defining or redefining the datum.

The term, reference, is used to indicate an identification of a procedure implying that the actions specified by the procedure will be made available.

A complete and rigorous discussion of reference and definition, including redefinition, is contained in Section 10.

5.1 Data and Procedure Names. A data name identifies a constant, a variable, an array, or an array element. A procedure name identifies a function or a subroutine.

5.1.1 *Constants.* A constant is a name that references a value. A constant may not be redefined.

An integer or real constant is said to be signed when it is immediately preceded by a plus or minus. Also, for these types, an optionally signed constant is either a constant or a signed constant.

5.1.1.1 *Integer Constant.* An integer constant is formed by a nonempty string of digits. The datum formed this way is interpreted as the value represented by the digit string.

5.1.1.2 *Real Constant.* A basic real constant is formed by an integer part, a decimal point, and a decimal fraction part in that order. Both the integer part and the decimal fraction part are formed by a string of digits; either one of these strings may be empty, but not both. The datum formed this way is interpreted as representing a value that is an approximation to the number represented by the integer and fraction parts.

A decimal exponent is formed by the letter E followed by an optionally signed integer constant. This exponent is interpreted as a multiplier (to be applied to the constant immediately preceding it) that is an approximation to ten raised to the power specified by the field following the E

A real constant is either a basic real constant or a basic real constant followed by a decimal exponent.

5.1.1.3

A double precision constant is a basic real constant followed by a double precision exponent or an integer constant followed by a double precision exponent.

5.1.1.4 *Complex Constant.* A complex constant is formed by an ordered pair of optionally signed real constants, separated by a comma, and enclosed within parentheses. The datum formed this way is interpreted as an approximation to the complex number represented by the pair.

5.1.1.5 *Logical Constant.* A logical constant is formed as one of the strings .TRUE. or .FALSE. ; these are interpreted as representing the truth values of true and false, respectively.

5.1.1.6 *Hollerith Constant.* A Hollerith constant is formed by an integer constant (whose value n is greater than zero) followed by the letter H , followed by exactly *n* characters. Any *n* characters capable of representation by the processor may follow the H . However, the differing character sets of different processors may cause the interpretation of these constants to vary. The character blank is significant in a Hollerith constant.

This constant form is only defined for use in the argument list of a CALL statement and in the data initialization statement.

5.1.2 *Variable.* A variable is a datum that is identified by a symbolic name (3.5). Such a datum may be referenced and defined.

5.1.3 *Array.* An array is an ordered set of data of one, two, or three dimensions. An array is identified by a symbolic name. Identification of the entire ordered set is achieved via use of the array name.

5.1.3.1 *Array Element.* An array element is one of the members of the set of data of an array. An array element is identified by immediately following the array name with a qualifier, called a subscript, which points to the particular element of the array.

An array element may be referenced and defined.

5.1.3.2 *Subscript.* A subscript is formed by a parenthesized list of subscript expressions. Each subscript expression is separated by a comma from its successor, if there is a successor. The number of subscript expressions must correspond to the declared dimensionality (7.2.1.1), except in an EQUIVALENCE statement (7.2.1.4). Following evaluation of all of the subscript expressions, the array element successor function (7.2.1.1.1) determines the identified array element.

5.1.3.3 *Subscript Expressions.* A subscript expression is formed from one of the following constructs:

$c*v+k$
$c*v-k$
$c*v$
$v+k$
$v-k$
v
k

where *c* and *k* are integer constants and *v* is an integer variable reference. See Section 6 for a discussion of evaluation of expressions and 10.2.8 and 10.3 for requirements that apply to the use of a variable in a subscript.

5.1.4 *Procedures.* A procedure (Section 8) is identified by a symbolic name. A procedure is a statement function, an intrinsic function, a basic external function, an external function, or an external subroutine. Statement functions, intrinsic functions, basic external functions, and external functions are referred to as functions or function procedures; external subroutines as subroutines or subroutine procedures.

A function supplies a result to be used at the point of reference; a subroutine does not. Functions are referenced in a manner different from subroutines.

5.2 FUNCTION REFERENCE. A function reference consists of the function name followed by an actual argument list enclosed in parentheses. If the list contains more than one argument, the arguments are separated by commas. The allowable forms of function arguments are given in Section 8.

See 10.2.1 for a discussion of requirements that apply to function references.

5.1.1.4

5.1.1.5

5.1.1.6

5.1.2 *Variable.* A variable is a datum that is identified by a symbolic name (3.5). Such a datum may be referenced and defined.

5.1.3 *Array.* An array is an ordered set of data of one or two dimensions. An array is identified by a symbolic name. Identification of the entire ordered set is achieved via use of the array name.

5.1.3.1 *Array Element.* An array element is one of the members of the set of data of an array. An array element is identified by immediately following the array name with a qualifier, called a subscript, which points to the particular element of the array.

An array element may be referenced and defined.

5.1.3.2 *Subscript.* A subscript is formed by a parenthesized list of subscript expressions. The subscript expressions are separated by a comma if two are present. The number of subscript expressions must correspond to the declared dimensionality (7.2.1.1), except in an EQUIVALENCE statement (7.2.1.4). Following evaluation of all of the subscript expressions, the array element successor function (7.2.1.1.1) determines the identified array element.

5.1.3.3 *Subscript Expressions.* A subscript expression is formed from one of the following constructs:

$c*v+k$
$c*v-k$
$c*v$
$v+k$
$v-k$
v
k

where *c* and *k* are integer constants and *v* is an integer variable reference. See Section 6 for a discussion of evaluation of expressions and 10.2.8 and 10.3 for requirements that apply to the use of a variable in a subscript.

5.1.4 *Procedures.* A procedure (Section 8) is identified by a symbolic name. A procedure is a statement function, an intrinsic function, a basic external function, an external function, or an external subroutine. Statement functions, intrinsic functions, basic external functions, and external functions are referred to as functions or function procedures; external subroutines as subroutines or subroutine procedures.

A function supplies a result to be used at the point of reference; a subroutine does not. Functions are referenced in a manner different from subroutines.

5.2 FUNCTION REFERENCE. A function reference consists of the function name followed by an actual argument list enclosed in parentheses. If the list contains more than one argument, the arguments are separated by commas. The allowable forms of function arguments are given in Section 8.

See 10.2.1 for a discussion of requirements that apply to function references.

5.3 Type Rules for Data and Procedure Identifiers. The type of a constant is implicit in its name.

There is no type associated with a symbolic name that identifies a subroutine or a block.

A symbolic name that identifies a variable, an array, or a statement function may have its type specified in a type-statement. In the absence of an explicit declaration, the type is implied by the first character of the name: I, J, K, L, M, and N imply type integer; any other letter implies type real.

A symbolic name that identifies an intrinsic function or a basic external function when it is used to identify this designated procedure, has a type associated with it as specified in Tables 3 and 4.

In the program unit in which an external function is referenced, its type definition is defined in the same manner as for a variable and an array. For a function subprogram, type is specified either implicitly by its name or explicitly in the FUNCTION statement.

The same type is associated with an array element as is associated with the array name.

5.4 Dummy Arguments. A dummy argument of an external procedure identifies a variable, array, subroutine, or external function.

When the use of an external function name is specified, the use of a dummy argument is permissible if an external function name will be associated with that dummy argument. (Section 8.)

When the use of an external subroutine name is specified, the use of a dummy argument is permissible if an external subroutine name will be associated with that dummy argument.

When the use of a variable or array element reference is specified, the use of a dummy argument is permissible if a value of the same type will be made available through argument association.

Unless specified otherwise, when the use of a variable, array, or array element name is specified, the use of a dummy argument is permissible provided that a proper association with an actual argument is made.

The process of argument association is discussed in Sections 8 and 10.

6. EXPRESSIONS

This section gives the formation and evaluation rules for arithmetic, relational, and logical expressions. A relational expression appears only within the context of logical expressions. An expression is formed from elements and operators. See 10.3 for a discussion of requirements that apply to the use of certain entities in expressions.

6.1 Arithmetic Expressions. An arithmetic expression is formed with arithmetic operators and arithmetic elements. Both the expression and its constituent elements identify values of one of the types integer, real, double precision, or complex. The arithmetic operators are:

Operator	*Representing*
+	Addition, positive value (zero + element)
−	Subtraction, negative value (zero − element)
*	Multiplication
/	Division
**	Exponentiation

The arithmetic elements are primary, factor, term, signed term, simple arithmetic expression, and arithmetic expression.

A primary is an arithmetic expression enclosed in parentheses, a constant, a variable reference, an array element reference, or a function reference.

A factor is a primary or a construct of the form

*primary**primary*

A term is a factor or a construct of one of the forms

term/factor

or

*term*term*

5.3 Type Rules for Data and Procedure Identifiers. The type of a constant is implicit in its name.

There is no type associated with a symbolic name that identifies a subroutine.

A symbolic name that identifies a variable, an array, or a statement function has a type implied by the first character of the name: I, J, K, L, M, and N imply type integer; any other letter implies type real.

A symbolic name that identifies an intrinsic function or a basic external function when it is used to identify this designated procedure, has a type associated with it as specified in Tables 3 and 4.

In the program unit in which an external function is referenced or defined, its type definition is defined in the same manner as for a variable and an array.

The same type is associated with an array element as is associated with the array name.

5.4 Dummy Arguments. A dummy argument of an external procedure identifies a variable or an array.

Unless specified otherwise, when the use of a variable, array, or array element name is specified, the use of a dummy argument is permissible provided that a proper association with an actual argument is made.

The process of argument association is discussed in Sections 8 and 10.

6. EXPRESSIONS

This section gives the formation and evaluation rules for arithmetic expressions. An expression is formed from elements and operators. See 10.3 for a discussion of requirements that apply to the use of certain entities in expressions.

6.1 Arithmetic Expressions. An arithmetic expression is formed with arithmetic operators and arithmetic elements. Both the expression and its constituent elements identify values of one of the types integer or real. The arithmetic operators are:

Operator	*Representing*
+	Addition, positive value (zero + element)
−	Subtraction, negative value (zero − element)
*	Multiplication
/	Division
**	Exponentiation

The arithmetic elements are primary, factor, term, signed term, simple arithmetic expression, and arithmetic expression.

A primary is an arithmetic expression enclosed in parentheses, a constant, a variable reference, an array element reference, or a function reference.

A factor is a primary or a construct of the form:

*primary**primary*

A term is a factor or a construct of one of the forms:

term/factor

or

*term*term*

A signed term is a term immediately preceded by + or −.

A simple arithmetic expression is a term or two simple arithmetic expressions separated by a + or −.

An arithmetic expression is a simple arithmetic expression or a signed term or either of the preceding forms immediately followed by a + or − immediately followed by a simple arithmetic expression.

A primary of any type may be exponentiated by an integer primary, and the resultant factor is of the same type as that of the element being exponentiated. A real or double precision primary may be exponentiated by a real or double precision primary, and the resultant factor is of type real if both primaries are of type real and otherwise of type double precision. These are the only cases for which use of the exponentiation operator is defined.

By use of the arithmetic operators other than exponentiation, any admissible element may be combined with another admissible element of the same type, and the resultant element is of the same type. Further, an admissible real element may be combined with an admissible double precision or complex element; the resultant element is of type double precision or complex, respectively.

6.2 Relational Expressions. A relational expression consists of two arithmetic expressions separated by a relational operator and will have the value true or false as the relation is true or false, respectively. One arithmetic expression may be of type real or double precision and the other of type real or double precision, or both arithmetic expressions may be of type integer. If a real expression and a double precision expression appear in a relational expression, the effect is the same as a similar relational expression. This similar expression contains a double precision zero as the right hand arithmetic expression and the difference of the two original expressions (in their original order) as the left. The relational operator is unchanged. The relational operators are:

Operator	*Representing*
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

6.3 Logical Expressions. A logical expression is formed with logical operators and logical elements and has the value true or false. The logical operators are:

Operator	*Representing*
.OR.	Logical disjunction
.AND.	Logical conjunction
.NOT.	Logical negation

The logical elements are logical primary, logical factor, logical term, and logical expression.

A logical primary is a logical expression enclosed in parentheses, a relational expression, a logical constant, a logical variable reference, a logical array element reference, or a logical function reference.

A logical factor is a logical primary or .NOT. followed by a logical primary.

A logical term is a logical factor or a construct of the form:

logical term .AND. logical term

A logical expression is a logical term or a construct of the form:

logical expression .OR. logical expression

6.4 Evaluation of Expressions. A part of an expression need be evaluated only if such action is necessary to establish the value of the expression. The rules for formation of expressions imply the binding strength of operators. It should be noted that the range of the subtraction operator is the term that immediately succeeds it. The evaluation may proceed according to any valid formation sequence.

A signed term is a term immediately preceded by + or −.

A simple arithmetic expression is a term or two simple arithmetic expressions separated by a + or −.

An arithmetic expression is a simple arithmetic expression or a signed term or either of the preceding forms immediately followed by a + or − immediately followed by a simple arithmetic expression.

A primary of any type may be exponentiated by an integer primary and the resultant factor is of the same type as that of the element being exponentiated. A real primary may be exponentiated by a real primary, and the resultant factor is of type real. These are the only cases for which use of the exponentiation operator is defined.

By use of the arithmetic operators other than exponentiation, any admissible element may be combined with another admissible element of the same type, and the resultant element is of the same type.

6.2

6.3

6.4 Evaluation of Expressions. A part of an expression need be evaluated only if such action is necessary to establish the value of the expression. The rules for formation of expressions imply the binding strength of operators. It should be noted that the range of the subtraction operator is the term that immediately succeeds it. The evaluation may proceed accordingly to any valid formation sequence.

TABLE 1. Rules for Assignment of *e* to *v*

If v Type Is	*And e Type Is*	*The Assignment Rule Is**
Integer	Integer	Assign
Integer	Real	Fix & Assign
Integer	Double Precision	Fix & Assign
Integer	Complex	P
Real	Integer	Float & Assign
Real	Real	Assign
Real	Double Precision	DP Evaluate & Real Assign
Real	Complex	P
Double Precision	Integer	DP Float & Assign
Double Precision	Real	DP Evaluate & Assign
Double Precision	Double Precision	Assign
Double Precision	Complex	P
Complex	Integer	P
Complex	Real	P
Complex	Double Precision	P
Complex	Complex	Assign

* Notes.

(1) P means prohibited combination.

(2) Assign means transmit the resulting value, without change, to the entity.

(3) Real Assign means transmit to the entity as much precision of the most significant part of the resulting value as a real datum can contain.

(4) DP Evaluate means evaluate the expression according to the rules of 6.1 (or any more precise rules) then DP Float.

(5) Fix means truncate any fractional part of the result and transform that value to the form of an integer datum.

(6) Float means transform the value to the form of a real datum.

(7) DP Float means transform the value to the form of a double precision datum, retaining in the process as much of the precision of the value as a double precision datum can contain.

7.1.2 *Control Statements.* There are eight types of control statements:

(1) GO TO statements.
(2) arithmetic IF statement.
(3) logical IF statement.
(4) CALL statement.
(5) RETURN statement.
(6) CONTINUE statement.
(7) program control statements.
(8) DO statement.

The statement labels used in a control statement must be associated with executable statements within the same program unit in which the control statement appears.

7.1.2.1 GO TO *Statements.* There are three types of GO TO statements:

(1) Unconditional GO TO statement.
(2) Assigned GO TO statement.
(3) Computed GO TO statement.

7.1.2.1.1 *Unconditional* GO TO *Statement.* An unconditional GO TO statement is of the form:

GO TO *k*

where *k* is a statement label.

Execution of this statement causes the statement identified by the statement label to be executed next.

TABLE 1. Rules for Assignment of *e* to *v*

If v Type Is	*And e Type Is*	*The Assignment Rule Is**
Integer	Integer	Assign
Integer	Real	Fix & Assign
Real	Integer	Float & Assign
Real	Real	Assign

* Notes.

(1) Assign means transmit the resulting value, without change, to the entity.

(2) Fix means truncate any fractional part of the result and transform that value to the form of an integer datum.

(3) Float means transform the value to the form of a real datum.

7.1.2 *Control Statements.* There are seven types of control statements:

(1) GO TO statements.
(2) Arithmetic IF statement.
(3) CALL statement.
(4) RETURN statement.
(5) CONTINUE statement.
(6) Program control statements.
(7) DO statement.

The statement labels used in a control statement must be associated with executable statements within the same program unit in which the control statement appears.

7.1.2.1 GO TO *Statements.* There are two types of GO TO statements:

(1) Unconditional GO TO statement.
(2) Computed GO TO statement.

7.1.2.1.1 *Unconditional* GO TO *Statement.* An unconditional GO TO statement is of the form:

GO TO *k*

where *k* is a statement label.

Execution of this statement causes the statement identified by the statement label to be executed next.

7.1.2.7 *Program Control Statements.* There are two types of program control statements:

(1) STOP statement.

(2) PAUSE statement.

7.1.2.7.1 STOP *Statement.* A STOP statement is of one of the forms:

STOP n
or
STOP

where n is an octal digit string of length from one to five.

Execution of this statement causes termination of execution of the executable program.

7.1.2.7.2 PAUSE *Statement.* A PAUSE statement is of one of the forms:

PAUSE n
or
PAUSE

where n is an octal digit string of length from one to five.

The inception of execution of this statement causes a cessation of execution of this executable program. Execution must be resumable. At the time of cessation of execution the octal digit string is accessible. The decision to resume execution is not under control of the program; but if execution is resumed, execution of the PAUSE statement is completed.

7.1.2.8 DO *Statement.* A DO statement is of one of the forms:

DO n i = m_1 , m_2 , m_3
or
DO n i = m_1 , m_2

where:

(1) n is the statement label of an executable statement. This statement, called the terminal statement of the associated DO, must physically follow and be in the same program unit as that DO statement. The terminal statement may not be a GO TO of any form, arithmetic IF, RETURN, STOP, PAUSE, or DO statement, nor a logical IF containing any of these forms.

(2) i is an integer variable name; this variable is called the control variable.

(3) m_1 , called the initial parameter; m_2 , called the terminal parameter; and m_3 , called the incrementation parameter, are each either an integer constant or integer variable reference. If the second form of the DO statement is used so that m_3 is not explicitly stated, a value of one is implied for the incrementation parameter. At time of execution of the DO statement, m_1 , m_2 , and m_3 must be greater than zero.

Associated with each DO statement is a range that is defined to be those executable statements from and including the first executable statement following the DO, to and including the terminal statement associated with the DO. A special situation occurs when the range of a DO contains another DO statement. In this case, the range of the contained DO must be a subset of the range of the containing DO.

A completely nested nest is a set of DO statements and their ranges, and any DO statements contained within their ranges, such that the first occurring terminal statement of any of those DO statements physically follows the last occurring DO statement and the first occurring DO statement of the set is not in the range of any DO statement.

A DO statement is used to define a loop. The action succeeding execution of a DO statement is described by the following five steps:

1. The control variable is assigned the value represented by the initial parameter. This value must be less than or equal to the value represented by the terminal parameter.

2. The range of the DO is executed.

3. If control reaches the terminal statement, and after execution of the terminal statement, the control variable of the most recently executed DO statement associated with the terminal statement is incremented by the value represented by the associated incrementation parameter.

7.1.2.7 *Program Control Statements.* There are two types of program control statements:

(1) STOP statement

(2) PAUSE statement

7.1.2.7.1 STOP *Statement.* A STOP statement is of one of the forms:

STOP n
or
STOP

where n is an octal digit string of length from one to four.

Execution of this statement causes termination of execution of the executable program.

7.1.2.7.2 PAUSE *Statement.* A PAUSE statement is of one of the forms:

PAUSE n
or
PAUSE

where n is an octal digit string of length from one to four.

The inception of execution of this statement causes a cessation of execution of this executable program. Execution must be resumable. At the time of cessation of execution, the octal digit string is accessible. The decision to resume execution is not under control of the program; but if execution is resumed, execution of the PAUSE statement is completed.

7.1.2.8 DO *Statement.* A DO statement is of one of the forms:

DO n i = m_1 , m_2 , m_3
or
DO n i = m_1 , m_2

where:

(1) n is the statement label of an executable statement. This statement, called the terminal statement of the associated DO, must physically follow and be in the same program unit as that DO statement. The terminal statement may not be a GO TO of any form, arithmetic IF, RETURN, STOP, PAUSE, or DO statement.

(2) i is an integer variable name; this variable is called the control variable.

(3) m_1 , called the initial parameter; m_2 , called the terminal parameter; and m_3 , called the incrementation parameter, are each either an integer constant or integer variable reference. If the second form of the DO statement is used so that m_3 is not explicitly stated, a value of one is implied for the incrementation parameter. At time of execution of the DO statement, m_1 , m_2 , and m_3 must be greater than zero.

Associated with each DO statement is a range that is defined to be those executable statements from and including the first executable statement following the DO, to and including the terminal statement associated with the DO. A special situation occurs when the range of a DO contains another DO statement. In this case, the range of the contained DO must be a subset of the range of the containing DO.

A DO statement is used to define a loop. The action succeeding execution of a DO statement is described by the following five steps:

1. The control variable is assigned the value represented by the initial parameter. This value must be less than or equal to the value represented by the terminal parameter.

2. The range of the DO is executed.

3. If control reaches the terminal statement, and after execution of the terminal statement, the control variable of the most recently executed DO statement associated with the terminal statement is incremented by the value represented by the associated incrementation parameter.

4. If the value of the control variable after incrementation is less than or equal to the value represented by the associated terminal parameter, the action as described starting at step 2 is repeated with the understanding that the range in question is that of the DO, the control variable of which was most recently incremented. If the value of the control variable is greater than the value represented by its associated terminal parameter, the DO is said to have been satisfied and the control variable becomes undefined.

5. At this point, if there were one or more other DO statements referring to the terminal statement in question, the control variable of the next most recently executed DO statement is incremented by the value represented by its associated incrementation parameter and the action as described in step 4 is repeated until all DO statements referring to the particular termination statement are satisfied, at which time the first executable statement following the terminal statement is executed.

Upon exiting from the range of a DO by execution of a GO TO statement or an arithmetic IF statement, that is, other than by satisfying the DO, the control variable of the DO is defined and is equal to the most recent value attained as defined in the foregoing.

A DO is said to have an extended range if both of the following conditions apply:

(1) There exists a GO TO statement or arithmetic IF statement within the range of the innermost DO of a completely nested nest that can cause control to pass out of that nest.

(2) There exists a GO TO statement or arithmetic IF statement not within the nest that, in the collection of all possible sequences of execution in the particular program unit could be executed after a statement of the type described in (1), and the execution of which could cause control to return into the range of the innermost DO of the completely nested nest.

If both of these conditions apply, the extended range is defined to be the set of all executable statements that may be executed between all pairs of control statements, the first of which satisfies the condition of (1) and the second of (2). The first of the pair is not included in the extended range; the second is. A GO TO statement or an arithmetic IF statement may not cause control to pass into the range of a DO unless it is being executed as part of the extended range of that particular DO. Further, the extended range of a DO may not contain a DO that has an extended range. When a procedure reference occurs in the range of a DO, the actions of that procedure are considered to be temporarily within that range, i.e., during the execution of that reference.

The control variable, initial parameter, terminal parameter, and incrementation parameter of a DO may not be redefined during the execution of the range or extended range of that DO.

If a statement is the terminal statement of more than one DO statement, the statement label of that terminal statement may not be used in any GO TO or arithmetic IF statement that occurs anywhere but in the range of the most deeply contained DO with that terminal statement.

7.1.3 *Input/Output Statements.* There are two types of input/output statements:

(1) READ and WRITE statements.

(2) Auxiliary Input/Output statements.

The first type consists of the statements that cause transfer of records of sequential files to and from internal storage, respectively. The second type consists of the BACKSPACE and REWIND statements that provide for positioning of such an external file, and ENDFILE, which provides for demarcation of such an external file.

In the following descriptions, u and f identify input/output units and format specifications, respectively. An input/output unit is identified by an integer value and u may be either an integer constant or an integer variable reference whose value then identifies the unit. The format specification is described in Section 7.2.3. Either the statement label of a FORMAT state-

4. If the value of the control variable after incrementation is less than or equal to the value represented by the associated terminal parameter, the action as described starting at step 2 is repeated with the understanding that the range in question is that of the DO, the control variable of which was most recently incremented. If the value of the control variable is greater than the value represented by its associated terminal parameter, the DO is said to have been satisfied and the control variable becomes undefined.

5. At this point, if there were one or more other DO statements referring to the terminal statement in question, the control variable of the next most recently executed DO statement is incremented by the value represented by its associated incrementation parameter and the action as described in step 4 is repeated until all DO statements referring to the particular termination statement are satisfied, at which time the first executable statement following the terminal statement is executed.

Upon exiting from the range of a DO by execution of a GO TO statement or an arithmetic IF statement, that is, other than by satisfying the DO, the control variable of the DO is defined and is equal to the most recent value attained as defined in the foregoing.

A GO TO statement or an arithmetic IF statement may not cause control to pass into the range of a DO from outside its range. When a procedure reference occurs in the range of a DO, the actions of that procedure are considered to be temporarily within that range, i.e., during the execution of that reference.

The control variable, initial parameter, terminal parameter, and incrementation parameters of a DO may not be redefined during the execution of the range of that DO.

If a statement is the terminal statement of more than one DO statement, the statement label of that terminal statement may not be used in any GO TO or arithmetic IF statement that occurs anywhere but in the range of the most deeply contained DO with that terminal statement.

7.1.3 *Input/Output Statements.* There are two types of input/output statements:

(1) READ and WRITE statements.

(2) Auxiliary input/output statements.

The first type consists of the statements that cause transfer of records of sequential files to and from internal storage, respectively. The second type consists of the BACKSPACE and REWIND statements that provide for positioning of such an external file, and ENDFILE, which provides for demarcation of such an external file.

In the following descriptions, u and f identify input/output units and format specifications, respectively. An input/output unit is identified by an integer value and u may be either an integer constant or an integer variable reference whose value then identifies the unit. The format specification is described in 7.2.3. The statement label of a FORMAT statement is represented by f. The ment or an array name may be represented by *f*. If a statement label, the identified statement must appear in the same program unit as the input/output statement. If an array name, it must conform to the specifications in 7.2.3.10.

A particular unit has a single sequential file associated with it. The most general case of such a unit has the following properties:

(1) If the unit contains one or more records, those records exist as a totally ordered set.

(2) There exists a unique position of the unit called its initial point. If a unit contains no records, that unit is positioned at its initial point. If the unit is at its initial point and contains records, the first record of the unit is defined as the next record.

(3) If a unit is not positioned at its initial point, there exists a unique preceding record associated with that position. The least of any records in the ordering described by (1) following this preceding record is defined as the next record of that position.

(4) Upon completion of execution of a WRITE or ENDFILE statement, there exist no records following the records created by that statement.

(5) When the next record is transmitted, the position of the unit is changed so that this next record becomes the preceding record.

If a unit does not provide for some of the properties given in the foregoing, certain statements that will be defined may not refer to that unit. The use of such a statement is not defined for that unit.

7.1.3.1 READ *and* WRITE *Statements*. The READ and WRITE statements specify transfer of information. Each such statement may include a list of the names of variables, arrays, and array elements. The named elements are assigned values on input and have their values transferred on output.

Records may be formatted or unformatted. A formatted record consists of a string of the characters that are permissible in Hollerith constants (5.1.1.6). The transfer of such a record requires that a format specification be referenced to supply the necessary positioning and conversion specifications (7.2.3). The number of records transferred by the execution of a formatted READ or WRITE is dependent upon the list and referenced format specification (7.2.3.4). An unformatted record consists of a string of values. When an unformatted or formatted READ statement is executed, the required records on the identified unit must be, respectively, unformatted or formatted records.

7.1.3.1.1 *Input/Output Lists*. The input list specifies the names of the variables and array elements to which values are assigned on input. The output list specifies the references to variables and array elements whose values are transmitted. The input and output lists are of the same form.

Lists are formed in the following manner. A simple list is a variable name, an array element name, or an array name, or two simple lists separated by a comma.

A list is a simple list, a simple list enclosed in parentheses, a DO-implied list, or two lists separated by a comma.

A DO-implied list is a list followed by a comma and a DO-implied specification, all enclosed in parentheses.

A DO-implied specification is of one of the forms:

$$i = m_1 , m_2 , m_3$$

or

$$i = m_1 , m_2$$

The elements i, m_1 , m_2 , and m_3 are as defined for the DO statement (7.1.2.8). The range of DO-implied specification is the list of the DO-implied list and, for input lists, i, m_1 , m_2 , and m_3 may appear, within that range, only in subscripts.

A variable name or array element name specifies itself. An array name specifies all of the array element names defined by the array declarator, and they are specified in the order given by the array element successor function (7.2.1.1.1).

identified statement must appear in the same program unit as the input/output statement.

A particular unit has a single sequential file associated with it. The most general case of such a unit has the following properties:

(1) If the unit contains one or more records, those records exist as a totally ordered set.

(2) There exists a unique position of the unit called its initial point. If a unit contains no records, that unit is positioned at its initial point. If the unit is at its initial point and contains records, the first record of the unit is defined as the next record.

(3) If a unit is not positioned at its initial point, there exists a unique preceding record associated with that position. The least of any records in the ordering described by (1) following this preceding record is defined as the next record of that position.

(4) Upon completion of execution of a WRITE or ENDFILE statement, there exist no records following the records created by that statement.

(5) When the next record is transmitted, the position of the unit is changed so that this next record becomes the preceding record.

If a unit does not provide for some of the properties given in the preceding, certain statements that will be defined mav not refer to that unit. The use of such a statement is not defined for that unit.

7.1.3.1 READ *and* WRITE *Statements*. The READ and WRITE statements specify transfer of information. Each such statement may include a list of the names of variables, arrays, and array elements. The named elements are assigned values on input and have their values transferred on output.

Records may be formatted or unformatted. A formatted record consists of a string of characters. The transfer of such a record requires that a format specification be referenced to supply the necessary positioning and conversion specifications (7.2.3). The number of records transferred by the execution of a formatted READ or WRITE is dependent upon the list and referenced format specification (7.2.3.4). An unformatted record consists of a string of values. When an unformatted or formatted READ statement is executed, the required records on the identified unit must be, respectively, unformatted or formatted records.

7.1.3.1.1 *Input/Output Lists*. The input list specifies the names of the variables and array elements to which values are assigned on input. The output list specifies the references to variables and array elements whose values are transmitted. The input and output lists are of the same form.

Lists are formed in the following manner. A simple list is a variable name, an array element name, or an array name, or two simple lists separated by a comma.

A list is a simple list, a simple list enclosed in parentheses, a DO-implied list, or two lists separated by commas.

A DO-implied list is a list followed by a comma and a DO-implied specification, all enclosed in parentheses.

A DO-implied specification is of one of the forms:

$$i = m_1 , m_2 , m_3$$

or

$$i = m_1 , m_2$$

The elements i, m_1 , m_2 , and m_3 are as defined for the DO statement (7.1.2.8). The range of DO-implied specification is the list of the DO-implied list and, for input lists, i, m_1, m_2, and m_3 may appear, within that range, only in subscripts.

A variable name or array element name specifies itself. An array name specifies all of the array element names defined by the array declarator, and they are specified in the order given by the array element successor function (7.2.1.1.1).

The elements of a list are specified in the order of their occurrence from left to right. The elements of a list in a DO-implied list are specified for each cycle of the implied DO.

7.1.3.1.2 *Formatted* READ. A formatted READ statement is of one of the forms:

READ (u, f) k

or

READ (u, f)

where k is a list.

Execution of this statement causes the input of the next records from the unit identified by u. The information is scanned and converted as specified by the format specification identified by f. The resulting values are assigned to the elements specified by the list. See however 7.2.3.4.

7.1.3.1.3 *Formatted* WRITE. A formatted WRITE statement is of one of the forms:

WRITE (u, f) k

or

WRITE (u, f)

where k is a list.

Execution of this statement creates the next records on the unit identified by u. The list specifies a sequence of values. These are converted and positioned as specified by the format specification identified by f. See however 7.2.3.4.

7.1.3.1.4 *Unformatted* READ. An unformatted READ statement is of one of the forms:

READ (u) k

or

READ (u)

where k is a list.

Execution of this statement causes the input of the next record from the unit identified by u, and, if there is a list, these values are assigned to the sequence of elements specified by the list. The sequence of values required by the list may not exceed the sequence of values from the unformatted record.

7.1.3.1.5 *Unformatted* WRITE. An unformatted WRITE statement is of the form:

WRITE (u) k

where k is a list.

Execution of this statement creates the next record on the unit identified by u of the sequence of values specified by the list.

7.1.3.2 *Auxiliary Input/Output Statements*. There are three types of auxiliary input/output statements:

(1) REWIND statement.
(2) BACKSPACE statement.
(3) ENDFILE statement.

7.1.3.2.1 REWIND *Statement*. A REWIND statement is of the form:

REWIND u

Execution of this statement causes the unit identified by u to be positioned at its initial point.

7.1.3.2.2 BACKSPACE *Statement*. A BACKSPACE statement is of the form:

BACKSPACE u

If the unit identified by u is positioned at its initial point, execution of this statement has no effect. Otherwise, the execution of this statement results in the positioning of the unit identified by u so that what had been the preceding record prior to that execution becomes the next record.

7.1.3.2.3 ENDFILE *Statement*. An ENDFILE statement is of the form:

ENDFILE u

Execution of this statement causes the recording of an endfile record on the unit identified by u. The endfile record is an unique record signifying a demarcation of a sequential file. Action is undefined when an endfile record is encountered during execution of a READ statement.

The elements of a list are specified in the order of their occurrence from left to right. The elements of a list in a DO-implied list are specified for each cycle of the implied DO.

7.1.3.1.2 *Formatted* READ. A formatted READ statement is of one of the forms:

READ $(u, f)k$

or

READ (u, f)

where k is a list.

Execution of this statement causes the input of the next records from the unit identified by u. The information is scanned and converted as specified by the format specification identified by f and the resulting values are assigned to the elements specified by the list. See however 7.2.3.4.

7.1.3.1.3 *Formatted* WRITE. A formatted WRITE statement is of one of the forms:

WRITE $(u, f)k$

or

WRITE (u, f)

where k is a list.

Execution of this statement creates the next records on the unit identified by u. The list specifies a sequence of values, and these are converted and positioned as specified by the format specification identified by f. See however 7.2.3.4.

7.1.3.1.4 *Unformatted* READ. An unformatted READ statement is of one of the forms:

READ $(u)k$

or

READ (u)

where k is a list.

Execution of this statement causes the input of the next record from the unit identified by u, and, if there is a list, these values are assigned to the sequence of elements specified by the list. The sequence of values required by the list may not exceed the sequence of values from the unformatted record.

7.1.3.1.5 *Unformatted* WRITE. An unformatted WRITE statement is of the form:

WRITE $(u)k$

where k is a list.

Execution of this statement creates the next record on the unit identified by u of the sequence of values specified by the list.

7.1.3.2 *Auxiliary Input/Output Statements*. There are three types of auxiliary input/output statements:

(1) REWIND statement.
(2) BACKSPACE statement.
(3) ENDFILE statement.

7.1.3.2.1 REWIND *Statement*. A REWIND statement is of the form:

REWIND u

Execution of this statement causes the unit identified by u to be positioned at its initial point.

7.1.3.2.2 BACKSPACE *Statement*. A BACKSPACE statement is of the form:

BACKSPACE u

If the unit identified by u is positioned at its initial point, execution of this statement has no.effect. Otherwise, the execution of this statement results in the positioning of the unit identified by u so that what had been the preceding record prior to that execution becomes the next record.

7.1.3.2.3 ENDFILE *Statement*. An ENDFILE statement is of the form:

ENDFILE u

Execution of this statement causes the recording of an endfile record on the unit identified by u. The endfile record is a unique record signifying a demarcation of a sequential file. Action is undefined when an endfile record is encountered during execution of a READ statement.

7.1.3.3 *Printing of Formatted Record.* When formatted records are prepared for printing, the first character of the record is not printed.

The first character of such a record determines vertical spacing as follows:

Character	*Vertical Spacing Before Printing*
Blank	One line
0	Two lines
1	To first line of next page
+	No advance

7.2 Nonexecutable Statements. There are five types of nonexecutable statements:

(1) Specification statements.
(2) Data initialization statement.
(3) FORMAT statement.
(4) Function defining statements.
(5) Subprogram statements.

See 10.1.2 for a discussion of restrictions on appearances of symbolic names in such statements.

The function defining statements and subprogram statements are discussed in Section 8.

7.2.1 *Specification Statements.* There are five types of specification statements:

(1) DIMENSION statement.
(2) COMMON statement.
(3) EQUIVALENCE statement.
(4) EXTERNAL statement.
(5) Type-statements.

7.2.1.1 *Array-Declarator.* An array declarator specifies an array used in a program unit.

The array declarator indicates the symbolic name, the number of dimensions (one, two, or three), and the size of each of the dimensions. The array declarator statement may be a type-statement, DIMENSION, or COMMON statement.

An array declarator has the form:

$$v\ (i)$$

where:

(1) v, called the declarator name, is a symbolic name,

(2) (i), called the declarator subscript, is composed of 1, 2, or 3 expressions, each of which may be an integer constant or an integer variable name. Each expression is separated by a comma from its successor if there are more than one of them. In the case where i contains no integer variable, i is called the constant declarator subscript.

The appearance of a declarator subscript in a declarator statement serves to inform the processor that the declarator name is an array name. The number of subscript expressions specified for the array indicates its dimensionality. The magnitude of the values given for the subscript expressions indicates the maximum value that the subscript may attain in any array element name.

No array element name may contain a subscript that, during execution of the executable program, assumes a value less than one or larger than the maximum length specified in the array declarator.

7.2.1.1.1 *Array Element Successor Function and Value of a Subscript.* For a given dimensionality, subscript declarator, and subscript, the value of a subscript pointing to an array element and the maximum value a subscript may attain is indicated in Table 2. A subscript expression must be greater than zero.

The value of the array element successor function is obtained by adding one to the entry in the subscript value column. Any array element whose subscript has this value is the successor to the original element. The last element of the array is the one whose subscript value is the maximum subscript value and has no successor element.

7.1.3.3 *Printing of Formatted Records.* When formatted records are prepared for printing, the first character of such a record is not printed.

7.2 Nonexecutable Statements. There are four types of nonexecutable statements:

(1) Specification statements.
(2) FORMAT statement.
(3) Function defining statements.
(4) Subprogram statements.

See 10.1.2 for a discussion of restrictions on appearances of symbolic names in such statements.

The function defining statements and subprogram statements are discussed in Section 8.

7.2.1 *Specification Statements.* There are three types of specification statements:

(1) DIMENSION statement.
(2) COMMON statement.
(3) EQUIVALENCE statement.

7.2.1.1 *Array Declarator.* An array declarator specifies an array used in a program unit.

The array declarator indicates the symbolic name, the number of dimensions (one or two), and the size of each of the dimensions. The array declarator statement is the DIMENSION statement.

An array declarator has the form:

$$v\ (i)$$

where:

(1) v, called the declarator name, is a symbolic name.

(2) (i), called the declarator subscript, is composed of an integer constant or two integer constants separated by a comma.

The appearance of a declarator subscript in a declarator statement serves to inform the processor that the declarator name is an array name. The number of subscript expressions specified for the array indicates its dimensionality. The magnitude of the values given for the subscript expressions indicates the maximum value that the subscript may attain in any array element name.

No array element name may contain a subscript that, during execution of the executable program, assumes a value less than one or larger than the maximum length specified in the array declarator.

7.2.1.1.1 *Array Element Successor Function and Value of a Subscript.* For a given dimensionality, subscript declarator, and subscript, the value of a subscript pointing to an array element and the maximum value a subscript may attain are indicated in Table 2. A subscript expression must be greater than zero.

The value of the array element successor function is obtained by adding one to the entry in the subscript value column. Any array element whose subscript has this value is the successor to the original element. The last element of the array is the one whose subscript value is the maximum subscript value and has no successor element.

TABLE 2. VALUE OF A SUBSCRIPT

Dimensionality	*Subscript Declarator*	*Subscript*	*Subscript Value*	*Maximum Subscript Value*
1	(A)	(a)	a	A
2	(A, B)	(a, b)	$a + A \cdot (b-1)$	$A \cdot B$
3	(A, B, C)	(a, b, c)	$a + A \cdot (b-1)$ $+ A \cdot B \cdot (c-1)$	$A \cdot B \cdot C$

NOTES. (1) a, b, and c are subscript expressions.
(2) A, B, and C are dimensions.

7.2.1.1.2 *Adjustable Dimension.* If any of the entries in a declarator subscript is an integer variable name, the array is called an adjustable array, and the variable names are called adjustable dimensions. Such an array may only appear in a subprogram. The dummy argument list of the subprogram must contain the array name and the integer variable names that represent the adjustable dimensions. The values of the actual arguments that represent array dimensions in the argument list of the reference must be defined (10.2) prior to calling the subprogram and may not be redefined or undefined during execution of the subprogram. The maximum size of the actual array may not be exceeded. For every array appearing in an executable program (9.1.6), there must be at least one constant array declarator associated through subprogram references.

In a subprogram, a symbolic name that appears in a COMMON statement may not identify an adjustable array.

7.2.1.2 DIMENSION *Statement.* A DIMENSION statement is of the form:

DIMENSION $v_1(i_1)$, $v_2(i_2)$, $\cdots$, $v_n(i_n)$

where each $v(i)$ is an array declarator.

7.2.1.3 COMMON *Statement.* A COMMON statement is of the form:

COMMON / x_1 / a_1 / $\cdots$ / x_n / a_n

where each a is a nonempty list of variable names, array names, or array declarators (no dummy arguments are permitted) and each x is a symbolic name or is empty. If x_1 is empty, the first two slashes are optional. Each x is a block name, a name that bears no relationship to any variable or array having the same name. This holds true for any such variable or array in the same or any other program unit. See 10.1.1 for a discussion of restrictions on uses of block names.

In any given COMMON statement, the entities occurring between block name x and the next block name (or the end of the statement if no block name follows) are declared to be in common block x. All entities from the beginning of the statement until the appearance of a block name, or all entities in the statement if no block name appears, are declared to be in blank or unlabeled common. Alternatively, the appearance of two slashes with no block name between them declares the entities that follow to be in blank common.

A given common block name may occur more than once in a COMMON statement or in a program unit. The processor will string together in a given common block all entities so assigned in the order of their appearance (10.1.2). The first element of an array will follow the immediately preceding entity, if one exists, and the last element of an array will immediately precede the next entity, if one exists.

The size of a common block in a program unit is the sum of the storage required for the elements introduced through COMMON and EQUIVALENCE statements. The sizes of labeled common blocks with the same label in the program units that comprise an executable program must be the same. The sizes of blank common in the various program units that are to be executed together need not be the same. Size is measured in terms of storage units (7.2.1.3.1).

TABLE 2. VALUE OF A SUBSCRIPT

Dimensionality	*Subscript Declarator*	*Subscript*	*Subscript Value*	*Maximum Subscript Value*
1	(A)	(a)	a	A
2	(A, B)	(a, b)	$a + A \cdot (b-1)$	$A \cdot B$

NOTES. (1) a and b are subscript expressions.
(2) A and B are dimensions.

7.2.1.1.2

7.2.1.2 DIMENSION *Statement.* A DIMENSION statement is of the form:

DIMENSION v_1 (i_1), v_2 (i_2), $\cdots$, v_n (i_n)

where each v (i) is an array declarator.

7.2.1.3 COMMON *Statement.* A COMMON statement is of the form:

COMMON a_1 , a_2 , $\cdots$, a_n

where each a is a variable name or an array name.

In any given COMMON statement, the entities occurring in the list of variable names are declared to be in common.

More than one COMMON statement may appear in a program unit. The processor will string together in common all entities so assigned in the order of their appearance. The first element of an array will follow the immediately preceding entity, if one exists, and the last element of an array will immediately precede the next entity if one exists.

The size of common in a program unit is the sum of the storage required for the elements introduced through COMMON and EQUIVALENCE statements. The size of common in the various program units that are to be executed together need not be the same. Size is measured in terms of storage units (7.2.1.3.1).

7.2.1.3.1 *Correspondence of Common Blocks.* If all of the program units of an executable program that contain any definition of a common block of a particular name define that block such that:

(1) There is identity in type for all entities defined in the corresponding position from the beginning of that block,

(2) If the block is labeled and the same number of entities is defined for the block, then the values in the corresponding positions (counted by the number of preceding storage units) are the same quantity in the executable program.

A double precision or a complex entity is counted as two logically consecutive storage units; a logical, real, or integer entity, as one storage unit.

Then for common blocks with the same number of storage units or blank common:

(1) In all program units which have defined the identical type to a given position (counted by the number of preceding storage units) references to that position refer to the same quantity.

(2) A correct reference is made to a particular position assuming a given type if the most recent value assignment to that position was of the same type.

7.2.1.4 EQUIVALENCE *Statement.* An EQUIVALENCE statement is of the form:

EQUIVALENCE (k_1), (k_2), $\cdots$, (k_n)

in which each k is a list of the form:

a_1 , a_2 , $\cdots$, a_m.

Each a is either a variable name or an array element name (not a dummy argument), the subscript of which contains only constants, and m is greater than or equal to two. The number of subscript expressions of an array element name must correspond in number to the dimensionality of the array declarator or must be one (the array element successor function defines a relation by which an array can be made equivalent to a one dimensional array of the same length).

The EQUIVALENCE statement is used to permit the sharing of storage by two or more entities. Each element in a given list is assigned the same storage (or part of the same storage) by the processor. The EQUIVALENCE statement should not be used to equate mathematically two or more entities. If a two storage unit entity is equivalenced to a one storage unit entity, the latter will share space with the first storage unit of the former.

The assignment of storage to variables and arrays declared directly in a COMMON statement is determined solely by consideration of their type and the COMMON and array declarator statements. Entities so declared are always assigned unique storage, contiguous in the order declared in the COMMON statement.

The effect of an EQUIVALENCE statement upon common assignment may be the lengthening of a common block; the only such lengthening permitted is that which extends a common block beyond the last assignment for that block made directly by a COMMON statement.

When two variables or array elements share storage because of the effects of EQUIVALENCE statements, the symbolic names of the variables or arrays in question may not both appear in COMMON statements in the same program unit.

Information contained in 7.2.1.1.1, 7.2.1.3.1, and the present section suffices to describe the possibilities of additional cases of sharing of storage between array elements and entities of common blocks. It is incorrect to cause either directly or indirectly a single storage unit to contain more than one element of the same array.

7.2.1.5 EXTERNAL *Statement.* An EXTERNAL statement is of the form:

EXTERNAL v_1, $v_{2;}$ . . . , v_n

where each v is an external procedure name.

Appearance of a name in an EXTERNAL statement declares that name to be an external procedure name. If an external pro-

7.2.1.3.1 *Correspondence of Common Blocks.* If all of the program units of an executable program that contain any definition of common define common such that there is identity in type for all entities defined in the corresponding position from the beginning of common; then the values in the corresponding positions are the same quantity in the executable program.

Each real or integer entity counts as one storage unit.

For common:

(1) In all program units that have defined the identical type to a given position (counted by the number of preceding storage units) references to that position refer to the same quantity.

(2) A correct reference is made to a particular position assuming a given type if the most recent value assignment to that position was of the same type.

7.2.1.4 EQUIVALENCE *Statement.* An EQUIVALENCE statement is of the form:

EQUIVALENCE (k_1), (k_2), $\cdots$, (k_n)

in which each k is a list of the form:

a_1 , a_2 , $\cdots$, a_m

Each a is either a variable name or an array element name (not a dummy argument), the subscript of which contains only constants, and m is greater than or equal to two. The number of subscript expressions of an array element name must correspond in number to the dimensionality of the array declarator or must be one (the array element successor function defines a relation by which an array can be made equivalent to a one dimensional array of the same length).

The EQUIVALENCE statement is used to permit the sharing of storage by two or more entities. Each element in a given list is assigned the same storage (or part of the same storage) by the processor. The EQUIVALENCE statement should not be used to equate mathematically two or more entities.

The assignment of storage to variables and arrays declared directly in a COMMON statement is determined solely by consideration of their type and the COMMON and array declarator statements. Entities so declared are always assigned unique storage, contiguous in the order declared in the COMMON statement.

The effect of an EQUIVALENCE statement upon common assignment may be the lengthening of common; the only such lengthening permitted is that which extends common beyond the last assignment for common made directly by a COMMON statement.

When two variables or array elements share storage because of the effects of EQUIVALENCE statements, the symbolic names of the variables or arrays in question may not both appear in COMMON statements in the same program unit.

Information contained in 7.2.1.1.1, 7.2.1.3.1, and the present section suffices to describe the possibilities of additional cases of sharing of storage between array elements and entities of common blocks. It is incorrect to cause either directly or indirectly a single storage unit to contain more than one element of the same array.

7.2.1.5

cedure name is used as an argument to another external procedure, it must appear in an EXTERNAL statement in the program unit in which it is so used.

7.2.1.6 *Type-statements.* A type-statement is of the form:

$$t\ v_1, v_2, \cdots, v_n$$

where t is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL, and each v is a variable name, an array name, a function name, or an array declarator.

A type-statement is used to override or confirm the implicit typing, to declare entities to be of type double precision, complex, or logical, and may supply dimension information.

The appearance of a symbolic name in a type-statement serves to inform the processor that it is of the specified data type for all appearances in the program unit.

7.2.2 *Data Initialization Statement.* A data initialization statement is of the form:

$$\mathrm{DATA}\ k_1\ /\ d_1\ /\ ,\ k_2\ /\ d_2\ /\ ,\ \cdots,\ k_n\ /\ d_n\ /$$

where:

(1) Each k is a list containing names of variables and array elements,

(2) Each d is a list of constants and optionally signed constants, any of which may be preceded by $j*$,

(3) j is an integer constant.

If a list contains more than one entry, the entries are separated by commas.

Dummy arguments may not appear in the list k. Any subscript expression must be an integer constant.

When the form $j*$ appears before a constant it indicates that the constant is to be specified j times. A Hollerith constant may appear in the list d.

A data initialization statement is used to define initial values of variables or array elements. There must be a one-to-one correspondence between the list-specified items and the constants. By this correspondence, the initial value is established.

An initially defined variable or array element may not be in blank common. A variable or array element in a labeled common block may be initially defined only in a block data subprogram.

7.2.3 FORMAT *Statement.* FORMAT statements are used in conjunction with the input/output of formatted records to provide conversion and editing information between the internal representation and the external character strings.

A FORMAT statement is of the form:

$$\mathrm{FORMAT}\ (q_1t_1z_1t_2z_2 \cdots t_nz_nq_2)$$

where:

(1) $(q_1t_1z_1t_2z_2 \cdots t_nz_nq_2)$ is the format specification.

(2) Each q is a series of slashes or is empty.

(3) Each t is a field descriptor or group of field descriptors.

(4) Each z is a field separator.

(5) n may be zero.

A FORMAT statement must be labeled.

7.2.3.1 *Field Descriptors.* The format field descriptors are of the forms:

$sr\mathrm{F}w.d$
$sr\mathrm{E}w.d$
$sr\mathrm{G}w.d$
$sr\mathrm{D}w.d$
$r\mathrm{I}w$
$r\mathrm{L}w$
$r\mathrm{A}w$
$n\mathrm{H}h_1h_2 \cdots h_n$
$n\mathrm{X}$

where:

(1) The letters F, E, G, D, I, L, A, H, and X indicate the manner of conversion and editing between the internal and external representations and are called the conversion codes.

(2) w and n are nonzero integer constants representing the width of the field in the external character string.

7.2.1.6

7.2.2

7.2.3 FORMAT *Statement.* FORMAT statements are used in conjunction with the input/output of formatted records to provide conversion and editing information between the internal representation and the external character strings.

A FORMAT statement is of the form:

$$\mathrm{FORMAT}\ (q_1t_1z_1t_2z_2 \cdots t_nz_nq_2)$$

where:

(1) $(q_1t_1z_1t_2z_2 \cdots t_nz_nq_2)$ is the format specification.

(2) Each q is a series of slashes or is empty.

(3) Each t is a field descriptor or group of field descriptors.

(4) Each z is a field separator.

(5) n may be zero.

A FORMAT statement must be labeled.

7.2.3.1 *Field Descriptors.* The format field descriptors are of the forms:

$r\mathrm{F}w.d$
$r\mathrm{E}w.d$
$r\mathrm{I}w$
$n\mathrm{H}h_1h_2 \cdots h_n$
$n\mathrm{X}$

where:

(1) The letters F, E, I, H, and X indicate the manner of conversion and editing between the internal and external represenations and are called the conversion codes.

(2) w and n are nonzero integer constants representing the width of the field in the external character string.

(3) d is an integer constant representing the number of digits in the fractional part of the external character string (except for G conversion code).

(4) r, the repeat count, is an optional nonzero integer constant indicating the number of times to repeat the succeeding basic field descriptor.

(5) s is optional and represents a scale factor designator.

(6) Each h is one of the characters capable of representation by the processor.

For all descriptors, the field width must be specified. For descriptors of the form $w.d$, the d must be specified, even if it is zero. Further, w must be greater than or equal to d.

The phrase *basic field descriptor* will be used to signify the field descriptor unmodified by s or r.

The internal representation of external fields will correspond to the internal representation of the corresponding type constants (4.2 and 5.1.1).

7.2.3.2 *Field Separators.* The format field separators are the slash and the comma. A series of slashes is also a field separator. The field descriptors or groups of field descriptors are separated by a field separator.

The slash is used not only to separate field descriptors, but to specify demarcation of formatted records. A formatted record is a string of characters. The lengths of the strings for a given external medium are dependent upon both the processor and the external medium.

The processing of the number of characters that can be contained in a record by an external medium does not of itself cause the introduction or inception of processing of the next record.

7.2.3.3 *Repeat Specifications.* Repetition of the field descriptors (except nH and nX) is accomplished by using the repeat count. If the input/output list warrants, the specified conversion will be interpreted repetitively up to the specified number of times.

Repetition of a group of field descriptors or field separators is accomplished by enclosing them within parentheses and optionally preceding the left parenthesis with an integer constant called the group repeat count indicating the number of times to interpret the enclosed grouping. If no group repeat count is specified, a group repeat count of one is assumed. This form of grouping is called a basic group.

A further grouping may be formed by enclosing field descriptors, field separators, or basic groups within parentheses. Again, a group repeat count may be specified. The parentheses enclosing the format specification are not considered as group delineating parentheses.

7.2.3.4 *Format Control Interaction with an Input/Output List.* The inception of execution of a formatted READ or formatted WRITE statement initiates format control. Each action of format control depends on information jointly provided respectively by the next element of the input/output list, if one exists, and the next field descriptor obtained from the format specification. If there is an input/output list, at least one field descriptor other than nH or nX must exist.

When a READ statement is executed under format control, one record is read when the format control is initiated, and thereafter additional records are read only as the format specification demands. Such action may not require more characters of a record than it contains.

When a WRITE statement is executed under format control, writing of a record occurs each time the format specification demands that a new record be started. Termination of format control causes writing of the current record.

Except for the effects of repeat counts, the format specification is interpreted from left to right.

To each I, F, E, G, D, A, or L basic descriptor interpreted in a format specification, there corresponds one element specified by the input/output list, except that a complex element requires the interpretation of two F, E, or G basic descriptors. To each

(3) d is an integer constant representing the number of digits in the fractional part of the external character string.

(4) r, the repeat count, is an optional nonzero integer constant indicating the number of times to repeat the succeeding basic field descriptor.

(5) Each h is one character.

For all descriptors, the field width must be specified. For descriptors of the form $w.d$, the d must be specified, even if it is zero. Further, w must be greater than or equal to d.

The phrase *basic field descriptor* will be used to signify the field descriptor unmodified by r.

The internal representation of external fields will correspond to the internal representation of the corresponding type constants (4.2 and 5.1.1).

7.2.3.2 *Field Separators.* The format field separators are the slash and the comma. A series of slashes is also a field separator. The field descriptors or groups of field descriptors are separated by a field separator.

The slash is used not only to separate field descriptors, but to specify demarcation of formatted records. A formatted record is a string of characters. The lengths of the strings for a given external medium are dependent upon both the processor and the external medium.

The processing of the number of characters that can be contained in a record by an external medium does not of itself cause the introduction or inception of processing of the next record.

7.2.3.3 *Repeat Specifications.* Repetition of the field descriptors (except nH and nX) is accomplished by using the repeat count. If the input/output list warrants, the specified conversion will be interpreted repetitively up to the specified number of times.

Repetition of a group of field descriptors or field separators is accomplished by enclosing them within parentheses and optionally preceding the left parenthesis with an integer constant called the group repeat count indicating the number of times to interpret the enclosed grouping. If no group repeat count is specified, a group repeat count of one is assumed. This form of grouping is called a basic group.

7.2.3.4 *Format Control Interaction with an Input/Output List.* The inception of execution of a formatted READ or formatted WRITE statement initiates format control. Each action of format control depends on information jointly provided respectively by the next element of the input/output list, if one exists, and the next field descriptor obtained from the format specification. If there is an input/output list, at least one field descriptor other than nH or nX must exist.

When a READ statement is executed under format control, one record is read when the format control is initiated, and thereafter additional records are read only as the format specification demands. Such action may not require more characters of a record than it contains.

When a WRITE statement is executed under format control, writing of a record occurs each time the format specification demands that a new record be started. Termination of format control causes writing of the current record.

Except for the effects of repeat counts, the format specification is interpreted from left to right.

To each I, F, or E basic descriptor interpreted in a format specification, there corresponds one element specified by the input/output list. To each H or X basic descriptor there is no corresponding element specified by the input/output list, and the format

H or X basic descriptor there is no corresponding element specified by the input/output list, and the format control communicates information directly with the record. Whenever a slash is encountered, the format specification demands that a new record start or the preceding record terminate. During a READ operation, any unprocessed characters of the current record will be skipped at the time of termination of format control or when a slash is encountered.

Whenever the format control encounters an I, F, E, G, D, A, or L basic descriptor in a format specification, it determines if there is a corresponding element specified by the input/output list. If there is such an element, it transmits appropriately converted information between the element and the record and proceeds. If there is no corresponding element, the format control terminates.

If, however, the format control proceeds to the last outer right parenthesis of the format specification, a test is made to determine if another list element is specified. If not, control terminates. However, if another list element is specified, the format control demands a new record start and control reverts to that group repeat specification terminated by the last preceding right parenthesis, or if none exists, then to the first left parenthesis of the format specification. Note, this action of itself has no effect on the scale factor.

7.2.3.5 *Scale Factor.* A scale factor designator is defined for use with the F, E, G, and D conversions and is of the form:

nP

where n, the scale factor, is an integer constant or minus followed by an integer constant.

When the format control is initiated, a scale factor of zero is established. Once a scale factor has been established, it applies to all subsequently interpreted F, E, G, and D field descriptors, until another scale factor is encountered, and then that scale factor is established.

7.2.3.5.1 *Scale Factor Effects.* The scale factor n affects the appropriate conversions in the following manner:

(1) For F, E, G, and D input conversions (provided no exponent exists in the external field) and F output conversions, the scale factor effect is as follows:

externally represented number equals internally represented number times the quantity ten raised to the nth power.

(2) For F, E, G, and D input, the scale factor has no effect if there is an exponent in the external field.

(3) For E and D output, the basic real constant part of the output quantity is multiplied by 10^n and the exponent is reduced by n.

(4) For G output, the effect of the scale factor is suspended unless the magnitude of the datum to be converted is outside the range that permits the effective use of F conversion. If the effective use of E conversion is required, the scale factor has the same effect as with E output.

7.2.3.6 *Numeric Conversions.* The numeric field descriptors I, F, E, G, and D are used to specify input/output of integer, real, double precision, and complex data.

(1) With all numeric input conversions, leading blanks are not significant and other blanks are zero. Plus signs may be omitted. A field of all blanks is considered to be zero.

(2) With the F, E, G, and D input conversions, a decimal point appearing in the input field overrides the decimal point specification supplied by the field descriptor.

(3) With all output conversions, the output field is right justified. If the number of characters produced by the conversion is smaller than the field width, leading blanks will be inserted in the output field.

(4) With all output conversions, the external representation of a negative value must be signed; a positive value may be signed.

control communicates information directly with the record. Whenever a slash is encountered, the format specification demands that a new record start or the preceding record terminate. During a READ operation, any unprocessed characters of the current record will be skipped at the time of termination of format control or when a slash is encountered.

Whenever the format control encounters an I, F, or E basic descriptor in a format specification, it determines if there is a corresponding element specified by the input/output list. If there is such an element, it transmits appropriately converted information between the element and the record and proceeds. If there is no corresponding element, the format control terminates.

If, however, the format control proceeds to the last outer right parenthesis of the format specification, a test is made to determine if another list element is specified. If not, control terminates. However, if another list element is specified, the format control demands a new record start and control reverts to that group repeat specification terminated by the last preceding right parenthesis, or if none exists, then to the first left parenthesis of the format specification.

7.2.3.5

7.2.3.5.1

7.2.3.6 *Numeric Conversions.* The numeric field descriptors I, F, and E are used to specify input/output of integer and real data.

(1) In numeric input fields blanks are permitted only to the left of the first nonblank character or between the sign of the field and the next nonblank character. Such blanks are treated as zero in conversion. Plus signs may be omitted. A field of all blanks is considered to be zero.

(2) With the F and E input conversions, a decimal point appearing in the input field overrides the specification supplied by the field descriptor.

(3) With all output conversions, the output field is right justified. If the number of characters produced by the conversion is smaller than the field width, leading blanks will be inserted in the output field.

(4) With all output conversions, the external representation of a negative value must be signed; a positive value may be signed.

(5) The number of characters produced by an output conversion must not exceed the field width.

7.2.3.6.1 *Integer Conversion.* The numeric field descriptor Iw indicates that the external field occupies w positions as an integer. The value of the list item appears, or is to appear, internally as an integer datum.

In the external input field, the character string must be in the form of an integer constant or signed integer constant (5.1.1.1), except for the interpretation of blanks (7.2.3.6).

The external output field consists of blanks, if necessary, followed by a minus if the value of the internal datum is negative, or an optional plus otherwise, followed by the magnitude of the internal value converted to an integer constant.

7.2.3.6.2 *Real Conversions.* There are three conversions available for use with real data: F, E, and G.

The numeric field descriptor F$w.d$ indicates that the external field occupies w positions, the fractional part of which consists of d digits. The value of the list item appears, or is to appear, internally as a real datum.

The basic form of the external input field consists of an optional sign, followed by a string of digits optionally containing a decimal point. The basic form may be followed by an exponent of one of the following forms:

(1) Signed integer constant.

(2) E followed by an integer constant.

(3) E followed by a signed integer constant.

(4) D followed by an integer constant.

(5) D followed by a signed integer constant.

An exponent containing D is equivalent to an exponent containing E.

The external output field consists of blanks, if necessary, followed by a minus if the internal value is negative, or an optional plus otherwise, followed by string of digits containing a decimal point representing the magnitude of the internal value, as modified by the established scale factor, rounded to d fractional digits.

The numeric field descriptor E$w.d$ indicates that the external field occupies w positions, the fractional part of which consists of d digits. The value of the list item appears, or is to appear, internally as a real datum.

The form of the external input field is the same as for the F conversion.

The standard form of the external output field for a scale factor of zero is[1]

$$\xi 0.x_1 \cdots x_d Y$$

where:

(1) $x_1 \cdots x_d$ are the d most significant rounded digits of the value of the data to be output.

(2) Y is of one of the forms:

$$\mathrm{E} \pm y_1y_2$$
or
$$\pm y_1y_2y_3$$

and has the significance of a decimal exponent (an alternative for the plus in the first of these forms is the character blank).

(3) The digit 0 in the aforementioned standard form may optionally be replaced by no character position.

(4) Each y is a digit.

The scale factor n controls the decimal normalization between the number part and the exponent part such that:

(1) If $n \leqq 0$, there will be exactly $-n$ leading zeros and $d+n$ significant digits after the decimal point.

(2) If $n > 0$, there will be exactly n significant digits to the left of the decimal point and $d-n+1$ to the right of the decimal point.

The numeric field descriptor G$w.d$ indicates that the external field occupies w positions with d significant digits. The

[1] ξ signifies no character position or minus in that position.

(5) The number of characters produced by an output conversion must not exceed the field width.

7.2.3.6.1 *Integer Conversion.* The numeric field descriptor Iw indicates that the external field occupies w positions as an integer. The value of the list item appears, or is to appear, internally as an integer datum.

In the external input field, the character string must be in the form of an integer constant or signed integer constant (5.1.1.1), except for the interpretation of blanks (7.2.3.6).

The external output field consists of blanks, if necessary, followed by a minus if the value of the internal datum is negative, or an optional plus otherwise, followed by the magnitude of the internal value converted to an integer constant.

7.2.3.6.2 *Real Conversions.* There are two conversions available for use with real data: F and E.

The numeric field descriptor F$w.d$ indicates that the external field occupies w positions, the fractional part of which consists of d digits. The value of the list item appears, or is to appear, internally as a real datum.

The external input field consists of an optional sign, followed by a string of digits optionally containing a decimal point.

The external output field consists of blanks, if necessary, followed by a minus if the internal value is negative, or an optional plus otherwise, followed by a string of digits containing a decimal point representing the magnitude, to d fractional digits, of the internal value.

The numeric field descriptor E$w.d$ indicates that the external field occupies w positions, the fractional part of which consists of d digits. The value of the list item appears, or is to appear, internally as a real datum.

The basic form of the external input field is the same as for the F conversion. The basic form may be followed by an exponent of one of the following forms:

(1) Signed integer constant.

(2) E followed by an integer constant.

(3) E followed by a signed integer constant.

The standard form of the external output field is[1]

$$\xi 0.x_1 \cdots x_d Y$$

where:

(1) $x_1 \cdots x_d$ are the d most significant digits of the value of the data to be output.

(2) Y is of the form:

$$\mathrm{E} \pm y_1y_2$$

and has the significance of a decimal exponent (an alternative for the plus in the first of these forms is the character blank).

(3) Each y is a digit.

[1] ξ signifies no character position or minus in that position.

value of the list item appears, or is to appear, internally as a real datum.

Input processing is the same as for the F conversion.

The method of representation in the external output string is a function of the magnitude of the real datum being converted. Let N be the magnitude of the internal datum. The following tabulation exhibits a correspondence between N and the equivalent method of conversion that will be effected:

Magnitude of Datum	*Equivalent Conversion Effected*
$0.1 \leqq N < 1$	F$(w-4).d$, 4X
$1 \leqq N < 10$	F$(w-4).(d-1)$, 4X
$\vdots$	$\vdots$
$10^{d-2} \leqq N < 10^{d-1}$	F$(w-4)$.1, 4X
$10^{d-1} \leqq N < 10^{d}$	F$(w-4)$.0, 4X
Otherwise	sE$w.d$

Note that the effect of the scale factor is suspended unless the magnitude of the datum to be converted is outside of the range that permits effective use of F conversion.

7.2.3.6.3 *Double Precision Conversion.* The numeric field descriptor D$w.d$ indicates that the external field occupies w positions, the fractional part of which consists of d digits. The value of the list item appears, or is to appear, internally as a double precision datum.

The basic form of the external input field is the same as for real conversions.

The external output field is the same as for the E conversion, except that the character D may replace the character E in the exponent.

7.2.3.6.4 *Complex Conversion.* Since a complex datum consists of a pair of separate real data, the conversion is specified by two successively interpreted real field descriptors. The first of these supplies the real part. The second supplies the imaginary part.

7.2.3.7 *Logical Conversion.* The logical field descriptor Lw indicates that the external field occupies w positions as a string of information as defined below. The list item appears, or is to appear, internally as a logical datum.

The external input field must consist of optional blanks followed by a T or F followed by optional characters, for true and false, respectively.

The external output field consists of $w-1$ blanks followed by a T or F as the value of the internal datum is true or false, respectively.

7.2.3.8 *Hollerith Field Descriptor.* Hollerith information may be transmitted by means of two field descriptors, nH and Aw :

(1) The nH descriptor causes Hollerith information to be read into, or written from, the n characters (including blanks) following the nH descriptor in the format specification itself.

(2) The Aw descriptor causes w Hollerith characters to be read into, or written from, a specified list element.

Let g be the number of characters representable in a single storage unit (7.2.1.3.1). If the field width specified for A input is greater than or equal to g, the rightmost g characters will be taken from the external input field. If the field width is less than g, the w characters will appear left justified with $w-g$ trailing blanks in the internal representation.

If the field width specified for A output is greater than g, the external output field will consist of $w-g$ blanks, followed by the g characters from the internal representation. If the field width is less than or equal to g, the external output field will consist of the leftmost w characters from the internal representation.

7.2.3.9 *Blank Field Descriptor.* The field descriptor for blanks is nX . On input, n characters of the external input record are skipped. On output, n blanks are inserted in the external output record.

7.2.3.6.3

7.2.3.6.4

7.2.3.7

7.2.3.8 *Hollerith Field Descriptor.* Hollerith information may be transmitted by means of the field descriptor nH.

The nH descriptor causes Hollerith information to be read into, or written from, the n characters (including blanks) following the nH descriptor in the format specification itself.

7.2.3.9 *Blank Field Descriptor.* The field descriptor for blanks is nX. On input, n characters of the external input record are skipped. On output, n blanks are inserted in the external output record.

7.2.3.10 *Format Specification in Arrays.* Any of the formatted input/output statements may contain an array name in place of the reference to a FORMAT statement label. At the time an array is referenced in such a manner, the first part of the information contained in the array, taken in the natural order, must constitute a valid format specification. There is no requirement on the information contained in the array following the right parenthesis that ends the format specification.

The format specification which is to be inserted in the array has the same form as that defined for a FORMAT statement; that is, begins with a left parenthesis and ends with a right parenthesis. An nH field descriptor may not be part of a format specification within an array.

8. PROCEDURES AND SUBPROGRAMS

There are four categories of procedures: statement functions, intrinsic functions, external functions, and external subroutines. The first three categories are referred to collectively as functions or function procedures; the last as subroutines or subroutine procedures. There are two categories of subprograms: procedure subprograms and specification subprograms. Function subprograms and subroutine subprograms are classified as procedure subprograms. Block data subprograms are classified as specification subprograms. Type rules for function procedures are given in 5.3.

8.1 Statement Functions. A statement function is defined internally to the program unit in which it is referenced. It is defined by a single statement similar in form to an arithmetic or logical assignment statement.

In a given program unit, all statement function definitions must precede the first executable statement of the program unit and must follow the specification statements, if any. The name of a statement function must not appear in an EXTERNAL statement, nor as a variable name or an array name in the same program unit.

8.1.1 *Defining Statement Functions.* A statement function is defined by a statement of the form:

$$f(a_1\ ,\ a_2\ ,\ \cdots\ ,\ a_n)\ =\ e$$

where f is the function name, e is an expression, and the relationship between f and e must conform to the assignment rules in 7.1.1.1 and 7.1.1.2. The a's are distinct variable names, called the *dummy arguments* of the function. Since these are dummy arguments, their names, which serve only to indicate type, number, and order of arguments, may be the same as variable names of the same type appearing elsewhere in the program unit.

Aside from the dummy arguments, the expression e may only contain:

(1) Non-Hollerith constants.
(2) Variable references.
(3) Intrinsic function references.
(4) References to previously defined statement functions.
(5) External function references.

8.1.2 *Referencing Statement Functions.* A statement function is referenced by using its reference (5.2) as a primary in an arithmetic or logical expression. The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments. An actual argument in a statement function reference may be any expression of the same type as the corresponding dummy argument.

Execution of a statement function reference results in an association (10.2.2) of actual argument values with the corresponding dummy arguments in the expression of the function definition, and an evaluation of the expression. Following this, the resultant value is made available to the expression that contained the function reference.

7.2.3.10

8. PROCEDURES AND SUBPROGRAMS

There are four categories of procedures: statement functions, intrinsic functions, external functions, and external subroutines. The first three categories are referred to collectively as functions or function procedures; the last as subroutines or subroutine procedures. Function subprograms and subroutine subprograms are classified as procedure subprograms. Type rules for function procedures are given in 5.3.

8.1 Statement Functions. A statement function is defined internally to the program unit in which it is referenced. It is defined by a single statement similar in form to an arithmetic assignment statement.

In a given program unit, all statement function definitions must precede the first executable statement of the program unit and must follow the specification statements, if any. The name of a statement function must not appear as a variable name or an array name in the same program unit.

8.1.1 *Defining Statement Functions.* A statement function is defined by a statement of the form:

$$f\ (a_1\ ,\ a_2\ ,\ \cdots\ ,\ a_n)\ =\ e$$

where f is the function name, e is an expression, and the relationship between f and e must conform to the assignment rules in 7.1.1.1. The a's are distinct variable names, called the *dummy arguments* of the function. Since these are dummy arguments, their names, which serve only to indicate type, number, and order of arguments, may be the same as variable names of the same type appearing elsewhere in the program unit.

Aside from the dummy arguments, the expression e may only contain:

(1) Constants.
(2) Variable references.
(3) Intrinsic function references.
(4) References to previously defined statement functions.
(5) External function references.

8.1.2 *Referencing Statement Functions.* A statement function is referenced by using its reference (5.2) as a primary in an arithmetic expression. The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments. An actual argument in a statement function reference may be any expression of the same type as the corresponding dummy argument.

Execution of a statement function reference results in an association (10.2.2.) of actual argument values with the corresponding dummy arguments in the expression of the function definition, and an evaluation of the expression. Following this, the resultant value is made available to the expression that contained the function reference.

8.2 Intrinsic Functions and Their Reference. The symbolic names of the intrinsic functions (see Table 3) are predefined to the processor and have a special meaning and type if the name satisfies the conditions of 10.1.7.

An intrinsic function is referenced by using its reference as a primary in an arithmetic or logical expression. The actual arguments, which constitute the argument list, must agree in type, number, and order with the specification in Table 3 and may be any expression of the specified type. The intrinsic functions AMOD, MOD, SIGN, ISIGN, and DSIGN are not defined when the value of the second argument is zero.

Execution of an intrinsic function reference results in the actions specified in Table 3 based on the values of the actual arguments. Following this, the resultant value is made available to the expression that contained the function reference.

8.3 External Functions. An external function is defined externally to the program unit that references it. An external function defined by Fortran statements headed by a FUNCTION statement is called a function subprogram.

8.3.1 *Defining Function Subprograms.* A FUNCTION statement is of the form:

$$t \text{ FUNCTION } f \ (a_1, a_2, \cdots, a_n)$$

where:

(1) t is either INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL, or is empty.

(2) f is the symbolic name of the function to be defined.

(3) The a's, called the dummy arguments, are each either a variable name, an array name, or an external procedure name.

Function subprograms are constructed as specified in 9.1.3 with the following restrictions:

(1) The symbolic name of the function must also appear as a variable name in the defining subprogram. During every execution of the subprogram, this variable must be defined and, once defined, may be referenced or redefined. The value of the variable at the time of execution of any RETURN statement in this subprogram is called the value of the function.

(2) The symbolic name of the function must not appear in any nonexecutable statement in this program unit, except as the symbolic name of the function in the FUNCTION statement.

(3) The symbolic names of the dummy arguments may not appear in an EQUIVALENCE, COMMON, or DATA statement in the function subprogram.

(4) The function subprogram may define or redefine one or more of its arguments so as to effectively return results in addition to the value of the function.

(5) The function subprogram may contain any statements except BLOCK DATA, SUBROUTINE, another FUNCTION statement, or any statement that directly or indirectly references the function being defined.

(6) The function subprogram must contain at least one RETURN statement.

8.3.2 *Referencing External Functions.* An external function is referenced by using its reference (5.2) as a primary in an arithmetic or logical expression. The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments in the defining program unit. An actual argument in an external function reference may be one of the following:

(1) A variable name.
(2) An array element name.
(3) An array name.
(4) Any other expression.
(5) The name of an external procedure.

If an actual argument is an external function name or a subroutine name, then the corresponding dummy argument must be used as an external function name or a subroutine name, respectively.

8.2 Intrinsic Functions and Their Reference. The symbolic names of the intrinsic functions (see Table 3) are predefined to the processor and have a special meaning and type if the name satisfies the conditions of 10.1.7.

An intrinsic function is referenced by using its reference as a primary in an arithmetic expression. The actual arguments, which constitute the argument list, must agree in type, number, and order with the specification in Table 3 and may be any expression of the specified type. The intrinsic functions SIGN and ISIGN are not defined when the value of the second argument is zero.

Execution of an intrinsic function reference results in the actions specified in Table 3 based on the values of the actual arguments. Following this, the resultant value is made available to the expression that contained the function reference.

8.3 External Functions. An external function is defined externally to the program unit that references it. An external function defined by Fortran statements headed by a FUNCTION statement is called a function subprogram.

8.3.1 *Defining Function Subprograms.* A FUNCTION statement is of the form:

$$\text{FUNCTION } f \ (a_1, a_2, \cdots, a_n)$$

where:

(1) f is the symbolic name of the function to be defined.

(2) The a's, called the dummy arguments, are each either a variable name or an array name.

Function subprograms are constructed as specified in 9.1.3 with the following restrictions:

(1) The symbolic name of the function must also appear as a variable name in the defining subprogram. During every execution of the subprogram, this variable must be defined and, once defined, may be referenced or redefined. The value of the variable at the time of execution of any RETURN statement in this subprogram is called the value of the function.

(2) The symbolic name of the function must not appear in any nonexecutable statement in this program unit, except as the symbolic name of the function in the FUNCTION statement.

(3) The symbolic names of the dummy arguments may not appear in an EQUIVALENCE or COMMON statement in the function subprogram.

(4) The function subprogram may not define or redefine any of its arguments nor any entity in common.

(5) The function subprogram may contain any statements except SUBROUTINE, another FUNCTION statement, or any statement that directly or indirectly references the function being defined.

(6) The function subprogram must contain at least one RETURN statement.

8.3.2 *Referencing External Functions.* An external function is referenced by using its reference (5.2) as a primary in an arithmetic expression. The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments in the defining program unit. An actual argument in an external function reference may be one of the following:

(1) A variable name.
(2) An array element name.
(3) An array name.
(4) Any other expression.

Execution of an external function reference as described in the foregoing, results in an association (10.2.2) of actual arguments with all appearances of dummy arguments in executable statements and function definition statements. If the actual argument

TABLE 3. Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Symbolic Name	Type of:	
				Argument	Function
Absolute Value	$\mid a \mid$	1	ABS IABS DABS	Real Integer Double	Real Integer Double
Truncation	Sign of a times largest integer $\leqq \mid a \mid$	1	AINT INT IDINT	Real Real Double	Real Integer Integer
Remaindering* (see note below)	$a_1 \pmod{a_2}$	2	AMOD MOD	Real Integer	Real Integer
Choosing Largest Value	Max $(a_1, a_2, \cdots)$	$\geqq 2$	AMAX0 AMAX1 MAX0 MAX1 DMAX1	Integer Real Integer Real Double	Real Real Integer Integer Double
Choosing Smallest Value	Min $(a_1, a_2, \cdots)$	$\geqq 2$	AMIN0 AMIN1 MIN0 MIN1 DMIN1	Integer Real Integer Real Double	Real Real Integer Integer Double
Float	Conversion from integer to real	1	FLOAT	Integer	Real
Fix	Conversion from real to integer	1	IFIX	Real	Integer
Transfer of Sign	Sign of a_2 times $\mid a_1 \mid$	2	SIGN ISIGN DSIGN	Real Integer Double	Real Integer Double
Positive Difference	$a_1 -$ Min (a_1, a_2)	2	DIM IDIM	Real Integer	Real Integer
Obtain Most Significant Part of Double Precision Argument		1	SNGL	Double	Real
Obtain Real Part of Complex Argument		1	REAL	Complex	Real
Obtain Imaginary Part of Complex Argument		1	AIMAG	Complex	Real
Express Single Precision Argument in Double Precision Form		1	DBLE	Real	Double
Express Two Real Arguments in Complex Form	$a_1 + a_2\sqrt{-1}$	2	CMPLX	Real	Complex
Obtain Conjugate of a Complex Argument		1	CONJG	Complex	Complex

* The function MOD or AMOD (a_1, a_2) is defined as $a_1 - [a_1/a_2]a_2$, where $[x]$ is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as x.

TABLE 3. Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Symbolic Name	Type of:	
				Argument	Function
Absolute Value	$\mid a \mid$	1	ABS IABS	Real Integer	Real Integer
Float	Conversion from integer to real	1	FLOAT	Integer	Real
Fix	Conversion from real to integer	1	IFIX	Real	Integer
Transfer of Sign	Sign of a_2 times a_1	2	SIGN ISIGN	Real Integer	Real Integer

If an actual argument corresponds to a dummy argument that is defined or redefined in the referenced subprogram, the actual argument must be a variable name, an array element name, or an array name. Execution of an external function reference as described in the foregoing, results in an association (10.2.2) of actual arguments with all appearances of dummy arguments in executable statements, function definition statements, and as is as specified in item (4) in the foregoing, this association is by value rather than by name. Following these associations, execution of the first executable statement of the defining subprogram is undertaken.

An actual argument that is an array element name containing variables in the subscript could in every case be replaced by the same array name with a constant subscript containing the same values as would be derived by computing the variable subscript just before the association of arguments took place.

If a dummy argument of an external function is an array name, corresponding actual argument must be an array name.

adjustable dimensions in the defining subprogram. If the actual argument is as specified in item (4) in the foregoing, this association is by value rather than by name. Following these associations, execution of the first executable statement of the defining subprogram is undertaken. An actual argument which is an array element name containing variables in the subscript could in every case be replaced by the same argument with a constant subscript containing the same values as would be derived by computing the variable subscript just before the association of arguments takes place.

If a dummy argument of an external function is an array name, the corresponding actual argument must be an array name or array element name (10.1.3).

If a function reference causes a dummy argument in the referenced function to become associated with another dummy argument in the same function or with an entity in common, a definition of either within the function is prohibited.

Unless it is a dummy argument, an external function is also referenced (in that it must be defined) by the appearance of its symbolic name in an EXTERNAL statement.

8.3.3 *Basic External Functions.* FORTRAN processors must supply the external functions listed in Table 4. Referencing of these functions is accomplished as described in (8.3.2). Arguments for which the result of these functions is not mathematically defined or is of type other than that specified are improper.

8.4 SUBROUTINE. An external subroutine is defined externally to the program unit that references it. An external subroutine defined by FORTRAN statements headed by a SUBROUTINE statement is called a subroutine subprogram.

TABLE 4. BASIC EXTERNAL FUNCTIONS

Basic External Function	*Definition*	*Number of Arguments*	*Symbolic Name*	*Type of:*	
				Argument	*Function*
Exponential	e^a	1	EXP	Real	Real
		1	DEXP	Double	Double
		1	CEXP	Complex	Complex
Natural Logarithm	$\log_e (a)$	1	ALOG	Real	Real
		1	DLOG	Double	Double
		1	CLOG	Complex	Complex
Common Logarithm	$\log_{10} (a)$	1	ALOG10	Real	Real
			DLOG10	Double	Double
Trigonometric Sine	$\sin (a)$	1	SIN	Real	Real
		1	DSIN	Double	Double
		1	CSIN	Complex	Complex
Trigonometric Cosine	$\cos (a)$	1	COS	Real	Real
		1	DCOS	Double	Double
		1	CCOS	Complex	Complex
Hyperbolic Tangent	$\tanh (a)$	1	TANH	Real	Real
Square Root	$(a)^{1/2}$	1	SQRT	Real	Real
		1	DSQRT	Double	Double
		1	CSQRT	Complex	Complex
Arctangent	$\arctan (a)$	1	ATAN	Real	Real
		1	DATAN	Double	Double
	$\arctan (a_1/a_2)$	2	ATAN2	Real	Real
		2	DATAN2	Double	Double
Remaindering*	$a_1 \pmod{a_2}$	2	DMOD	Double	Double
Modulus		1	CABS	Complex	Real

* The function DMOD (a_1, a_2) is defined as $a_1 - [a_1/a_2]a_2$, where $[x]$ is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as the sign of x.

8.3.3 *Basic External Functions.* FORTRAN processors must supply the external functions listed in Table 4. Referencing of these functions is accomplished as described in 8.3.2. Arguments for which the result of these functions is not mathematically defined or is of type other than that specified are improper.

8.4 SUBROUTINE. An external subroutine is defined externally to the program unit that references it. An external subroutine defined by FORTRAN statements headed by a SUBROUTINE statement is called a subroutine subprogram.

TABLE 4. BASIC EXTERNAL FUNCTIONS

Basic External Function	*Definition*	*Number of Arguments*	*Symbolic Name*	*Type of:*	
				Argument	*Function*
Exponential	e^a	1	EXP	Real	Real
Natural logarithm	$\log_e (a)$	1	ALOG	Real	Real
Trigonometric sine	sine (a)	1	SIN	Real	Real
Trignometric cosine	$\cos (a)$	1	COS	Real	Real
Hyperbolic tangent	$\tanh (a)$	1	TANH	Real	Real
Square Root	$(a)^{1/2}$	1	SQRT	Real	Real
Arctangent	$\arctan(a)$	1	ATAN	Real	Real

8.4.1 *Defining Subroutine Subprograms.* A SUBROUTINE statement is of one of the forms:

SUBROUTINE s $(a_1, a_2, \cdots, a_n)$

or

SUBROUTINE s

where:

(1) s is the symbolic name of the subroutine to be defined.

(2) The a's, called the dummy arguments, are each either a variable name, an array name, or an external procedure name.

Subroutine subprograms are constructed as specified in 9.1.3 with the following restrictions:

(1) The symbolic name of the subroutine must not appear in any statement in this subprogram except as the symbolic name of the subroutine in the SUBROUTINE statement itself.

(2) The symbolic names of the dummy arguments may not appear in an EQUIVALENCE, COMMON, or DATA statement in the subprogram.

(3) The subroutine subprogram may define or redefine one or more of its arguments so as to effectively return results.

(4) The subroutine subprogram may contain any statements except BLOCK DATA, FUNCTION, another SUBROUTINE statement, or any statement that directly or indirectly references the subroutine being defined.

(5) The subroutine subprogram must contain at least one RETURN statement.

8.4.2 *Referencing Subroutines.* A subroutine is referenced by a CALL statement (7.1.2.4). The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments in the defining program. The use of a Hollerith constant as an actual argument is an exception to the rule requiring agreement of type. An actual argument in a subroutine reference may be one of the following:

(1) A Hollerith constant.
(2) A variable name.
(3) An array element name.
(4) An array name.
(5) Any other expression.
(6) The name of an external procedure.

If an actual argument is an external function name or a subroutine name, the corresponding dummy argument must be used as an external function name or a subroutine name, respectively.

If an actual argument corresponds to a dummy argument that is defined or redefined in the referenced subprogram, the actual argument must be a variable name, an array element name, or an array name.

Execution of a subroutine reference as described in the foregoing results in an association of actual arguments with all appearances of dummy arguments in executable statements, function definition statements, and as adjustable dimensions in the defining subprogram. If the actual argument is as specified in item (5) in the foregoing, this association is by value rather than by name. Following these associations, execution of the first executable statement of the defining subprogram is undertaken.

An actual argument which is an array element name containing variables in the subscript could in every case be replaced by the same argument with a constant subscript containing the same values as would be derived by computing the variable subscript just before the association of arguments takes place.

If a dummy argument of an external function is an array name, the corresponding actual argument must be an array name or array element name (10.1.3).

If a subroutine reference causes a dummy argument in the referenced subroutine to become associated with another dummy argument in the same subroutine or with an entity in common, a definition of either entity within the subroutine is prohibited.

Unless it is a dummy argument, a subroutine is also referenced (in that it must be defined) by the appearance of its symbolic name in an EXTERNAL statement.

8.4.1 *Defining Subroutine Subprograms.* A SUBROUTINE statement is of one of the forms:

SUBROUTINE $s(a_1, a_2, \cdots, a_n)$

or

SUBROUTINE s

where:

(1) s is the symbolic name of the subroutine to be defined.

(2) The a's, called the dummy arguments, are each either a variable name or an array name.

Subroutine subprograms are constructed as specified in 9.1.3 with the following restrictions:

(1) The symbolic name of the subroutine must not appear in any statement in this subprogram except as the symbolic name of the subroutine in the SUBROUTINE statement itself.

(2) The symbolic names of the dummy arguments may not appear in an EQUIVALENCE or COMMON statement in the subprogram.

(3) The subroutine subprogram may define or redefine one or more of its arguments so as to effectively return results.

(4) The subroutine subprogram may contain any statements except FUNCTION, another SUBROUTINE statement, or any statement that directly or indirectly references the subroutine being defined.

(5) The subroutine subprogram must contain at least one RETURN statement.

8.4.2 *Referencing Subroutines.* A subroutine is referenced by a CALL statement (7.1.2.4). The actual arguments, which constitute the argument list, must agree in order, number, and type with the corresponding dummy arguments in the defining program. An actual argument in a subroutine reference may be one of the following:

(1) A variable name.
(2) An array element name.
(3) An array name.
(4) Any other expression.

If an actual argument corresponds to a dummy argument that is defined or redefined in the referenced subprogram, the actual argument must be a variable name, an array element name, or an array name.

Execution of a subroutine reference as described in the foregoing results in an association of actual arguments with all appearances of dummy arguments in executable statements or function definition statements. If the actual argument is as specified in item (4) in the foregoing, this association is by value rather than by name. Following these associations, execution of the first executable statement of the defining subprogram is undertaken.

An actual argument that is an array element name containing variables in the subscript could, in every case, be replaced by the same array element name with a constant subscript containing the same values as would be derived by computing the variable subscript just before argument association took place.

If a dummy argument of an external function is an array name, the corresponding actual argument must be an array name.

If a subroutine reference causes a dummy argument in the referenced subroutine to become associated with another dummy argument in the same subroutine or with an entity in common, a definition of either entity within the subroutine is prohibited.

8.5 Block Data Subprogram. A BLOCK DATA statement is of the form:

BLOCK DATA

This statement may only appear as the first statement of specification subprograms that are called block data subprograms, and that are used to enter initial values into elements of labeled common blocks. This special subprogram contains only type-statements, EQUIVALENCE, DATA, DIMENSION, and COMMON statements.

If any entity of a given common block is being given an initial value in such a subprogram, a complete set of specification statements for the entire block must be included, even though some of the elements of the block do not appear in DATA statements. Initial values may be entered into more than one block in a single subprogram.

9. PROGRAMS

An executable program is a collection of statements, comment lines, and end lines that completely (except for input data values and their effects) describe a computing procedure.

9.1 Program Components. Programs consist of program parts, program bodies, and subprogram statements.

9.1.1 *Program Part.* A program part must contain at least one executable statement and may contain FORMAT statements, and data initialization statements. It need not contain any statements from either of the latter two classes of statement. This collection of statements may optionally be preceded by statement function definitions, data initialization statements, and FORMAT statements. As before only some or none of these need be present.

9.1.2 *Program Body.* A program body is a collection of specification statements, FORMAT statements or both, or neither, followed by a program part, followed by an end line.

9.1.3 *Subprogram.* A subprogram consists of a SUBROUTINE or FUNCTION statement followed by a program body, or is a block data subprogram.

9.1.4 *Block Data Subprogram.* A block data subprogram consists of a BLOCK DATA statement, followed by the appropriate (8.5) specification statements, followed by data initialization statements, followed by an end line.

9.1.5 *Main Program.* A main program consists of a program body.

9.1.6 *Executable Program.* An executable program consists of a main program plus any number of subprograms, external procedures, or both.

9.1.7 *Program Unit.* A program unit is a main program or a subprogram.

9.2 Normal Execution Sequence. When an executable program begins operation, execution commences with the execution of the first executable statement of the main program. A subprogram, when referenced, starts execution with execution of the first executable statement of that subprogram. Unless a statement is a GO TO, arithmetic IF, RETURN, or STOP statement or the terminal statement of a DO, completion of execution of that statement causes execution of the next following executable statement. The sequence of execution following execution of any of these statements is described in Section 7. A program part may not contain an executable statement that can never be executed.

A program part must contain a first executable statement.

10. INTRA- AND INTERPROGRAM RELATIONSHIPS

10.1 Symbolic Names. A symbolic name has been defined to consist of from one to six alphanumeric characters, the first of which must be alphabetic. Sequences of characters that are format field descriptors or uniquely identify certain statement types, e.g., GO TO, READ, FORMAT, etc. are not symbolic names in such occurrences nor do they form the first characters of symbolic

8.5

9. PROGRAMS

An executable program is a collection of statements, comment lines, and end lines that completely (except for input data values and their effects) describe a computing procedure.

9.1 Program Components. Programs consist of program parts, program bodies, and subprogram statements.

9.1.1 *Program Part.* A program part must contain at least one executable statement and may but need not contain FORMAT statements.

9.1.2 *Program Body.* A program body is a collection of optional specification statements optionally followed by statement function definitions, followed by a program part, followed by an end line. The specification statements must be in the following order: DIMENSION, COMMON, and EQUIVALENCE.

9.1.3 *Subprogram.* A subprogram consists of a SUBROUTINE or FUNCTION statement followed by a program body.

9.1.4

9.1.5 *Main Program.* A main program consists of a program body.

9.1.6 *Executable Program.* An executable program consists of a main program plus any number of subprograms, external procedures, or both.

9.1.7 *Program Unit.* A program unit is a main program or a subprogram.

9.2 Normal Execution Sequence. When an executable program begins operation, execution commences with the execution of the first executable statement of the main program. A subprogram, when referenced, starts execution with execution of the first executable statement of that subprogram. Unless a statement is a GO TO, arithmetic IF, RETURN, or STOP statement or the terminal statement of a DO, completion of execution of that statement causes execution of the next following executable statement. The sequence of execution following execution of any of these statements is described in Section 7. A program part may not contain an executable statement that can never be executed.

A program part must contain a first executable statement.

10. INTRA- AND INTERPROGRAM RELATIONSHIPS

10.1 Symbolic Names. A symbolic name has been defined to consist of from one to five alphanumeric characters, the first of which must be alphabetic. Sequences of characters that are format field descriptors or uniquely identify certain statement types, e.g., GO TO, READ, etc., are not symbolic names in such occurrences nor do they form the first characters of symbolic names in names in these cases. In a program unit, a symbolic name (perhaps qualified by a subscript) must identify an element of one (and usually only one) of the following classes:

Class I An array and the elements of that array.
Class II A variable.
Class III A statement function.
Class IV An intrinsic function.
Class V An external function.
Class VI A subroutine.
Class VII An external procedure which cannot be classified as either a subroutine or an external function in the program unit in question.
Class VIII A block name.

10.1.1 *Restrictions on Class.* A symbolic name in Class VIII in a program unit may also be in any one of the Classes I, II, or III in that program unit.

In the program unit in which a symbolic name in Class V appears immediately following the word FUNCTION in a FUNCTION statement, that name must also be in Class II.

Once a symbolic name is used in Class V, VI, VII, or VIII in any unit of an executable program, no other program unit of that executable program may use that name to identify an entity of these classes other than the one originally identified. In the totality of the program units that make up an executable program, a Class VII name must be associated with a Class V or VI name. Class VII can only exist locally in program units.

In a program unit, no symbolic name can be in more than one class except as noted in the foregoing. There are no restrictions on uses of symbolic names in different program units of an executable program other than those noted in the foregoing.

10.1.2 *Implications of Mentions in Specification and* DATA *Statements.* A symbolic name is in Class I if and only if it appears as a declarator name. Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in a COMMON statement (other than as a block name) is either in Class I, or in Class II but not Class V. (8.3.1) Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in an EQUIVALENCE statement is either in Class I, or in Class II but not Class V. (8.3.1)

A symbolic name that appears in a type-statement cannot be in Class VI or Class VII. Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in an EXTERNAL statement is in either Class V, Class VI, or Class VII. Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in a DATA statement is in either Class I, or in Class II but not Class V. (8.3.1) In an executable program, a storage unit (7.2.1.3.1) may have its value initialized one time at the most.

10.1.3 *Array and Array Element.* In a program unit, any appearance of a symbolic name that identifies an array must be immediately followed by a subscript, except for the following cases:

(1) In the list of an input/output statement.
(2) In a list of dummy arguments.
(3) In the list of actual arguments in a reference to an external procedure.
(4) In a COMMON statement.
(5) In a type-statement.

Only when an actual argument of an external procedure reference is an array name or an array element name may the corresponding dummy argument be an array name. If the actual argument is an array name, the length of the dummy argument array must be no greater than the length of the actual argument array. If the actual argument is an array element name, the length of the dummy argument array must be less than or equal to the length of the actual argument array plus one minus the value of the subscript of the array element.

these cases. In a program unit, a symbolic name (perhaps qualified by a subscript) must identify an element of one (and usually only one) of the following classes:

Class I An array and the elements of that array.
Class II A variable.
Class III A statement function.
Class IV An intrinsic function.
Class V An external function.
Class VI A subroutine.

10.1.1 *Restrictions on Class.* In the program unit in which a symbolic name in Class V appears immediately following the word FUNCTION in a FUNCTION statement, that name must also be in Class II.

Once a symbolic name is used in Class V or VI in any unit of an executable program, no other program unit of that executable program may use that name to identify an entity of these classes other than the one originally identified.

In a program unit, no symbolic name can be in more than one class except as noted in the foregoing. There are no restrictions on uses of symbolic names in different program units of an executable program other than those noted in the foregoing.

10.1.2 *Implications of Mentions in Specification Statements.* A symbolic name is in Class I if it appears as a declarator name and is not in Class III. Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in a COMMON statement is either in Class I, or in Class II but not Class V (8.3.1). Only one such appearance for a symbolic name in a program unit is permitted.

A symbolic name that appears in an EQUIVALENCE statement is either in Class I, or in Class II but not Class V (8.3.1).

10.1.3 *Array and Array Element.* In a program unit, any appearance of a symbolic name that identifies an array must be immediately followed by a subscript, except for the following cases:

(1) In the list of an input/output statement.
(2) In a list of dummy arguments.
(3) In the list of actual arguments in a reference to an external procedure.
(4) In a COMMON statement.

Only when an actual argument of an external procedure reference is an array name may the corresponding dummy argument be an array name. If the actual argument is an array name, the length of the dummy argument array must agree with the length of the actual argument array.

10.1.4 *External Procedures.* The only case when a symbolic name is in Class VII occurs when that name appears only in an EXTERNAL statement and as an actual argument to an external procedure in a program unit.

Only when an actual argument of an external procedure reference is an external procedure name may the corresponding dummy argument be an external procedure name.

In the execution of an executable program, a procedure subprogram may not be referenced twice without the execution of a RETURN statement in that procedure having intervened.

10.1.5 *Subroutine.* A symbolic name is in Class VI if it appears:

(1) Immediately following the word SUBROUTINE in a SUBROUTINE statement.

(2) Immediately following the word CALL in a CALL statement.

10.1.6 *Statement Function.* A symbolic name is in Class III in a program unit if and only if it meets all three of the following conditions:

(1) It does not appear in an EXTERNAL statement nor is it in Class I.

(2) Every appearance of the name, except in a type-statement, is immediately followed by a left parenthesis.

(3) A function defining statement (8.1.1) is present for that symbolic name.

10.1.7 *Intrinsic Function.* A symbolic name is in Class IV in a program unit if and only if it meets all four of the following conditions:

(1) It does not appear in an EXTERNAL statement nor is it in Class I or Class III.

(2) The symbolic name appears in the name column of the table in Section 8.2.

(3) The symbolic name does not appear in a type-statement of type different from the intrinsic type specified in the table.

(4) Every appearance of the symbolic name (except in a type-statement as described in the foregoing) is immediately followed by an actual argument list enclosed in parentheses.

The use of an intrinsic function in a program unit of an executable program does not preclude the use of the same symbolic name to identify some other entity in a different program unit of that executable program.

10.1.8 *External Function.* A symbolic name is in Class V if it:

(1) Appears immediately following the word FUNCTION in a FUNCTION statement

(2) Is not in Class I, Class III, Class IV, or Class VI and appears immediately followed by a left parenthesis on every occurrence except in a type-statement, in an EXTERNAL statement, or as an actual argument. There must be at least one such appearance in the program unit in which it is so used.

10.1.9 *Variable.* In a program unit, a symbolic name is in Class II if it meets all three of the following conditions:

(1) It is not in Class VI or Class VII.

(2) It is never immediately followed by a left parenthesis unless it is immediately preceded by the word FUNCTION in a FUNCTION statement.

(3) It occurs other than in a Class VIII appearance.

10.1.10 *Block Name.* A symbolic name is in Class VIII if and only if it is used as a block name in a COMMON statement.

10.2 Definition. There are two levels of definition of numeric values, first level definition and second level definition. The concept of definition on the first level applies to array elements and variables; that of second level definition to integer variables only. These concepts are defined in terms of progression of execution; and thus, an executable program, complete and in execution, is assumed in what follows.

There are two other varieties of definition that should be noted. The first, effected by GO TO assignment and referring to an integer variable being defined with other than an integer value, is

10.1.4 *External Procedures.* In the execution of an executable program, a procedure subprogram may not be referenced twice without the execution of a RETURN statement in that procedure having intervened.

10.1.5 *Subroutine.* A symbolic name is in Class VI if it appears:

(1) Immediately following the word SUBROUTINE in a SUBROUTINE statement.

(2) Immediately following the word CALL in a CALL statement.

10.1.6. *Statement Function.* A symbolic name is in Class III in a program unit if and only if it meets all three of the following conditions:

(1) It is not in Class I or Class IV.

(2) Every appearance of the name is immediately followed by a left parenthesis.

(3) A function defining statement is present for that symbolic name.

10.1.7 *Intrinsic Function.* A symbolic name is in Class IV in a program unit if and only if it meets both of the following conditions:

(1) The symbolic name appears in the name column of Table 3.

(2) Every appearance of the symbolic name is immediately followed by an actual argument list enclosed in parentheses.

The use of an intrinsic function in a program unit of an executable program does not preclude the use of the same symbolic name to identify some other entity in a different program unit of that executable program.

10.1.8 *External Function.* A symbolic name is in Class V if it:

(1) Appears immediately following the word FUNCTION in a FUNCTION statement.

(2) Is not in Class I, Class III, Class IV, or Class VI and appears immediately followed by a left parenthesis on every occurrence. There must be at least one such appearance in the program unit in which it is so used.

10.1.9 *Variable.* In a program unit, a symbolic name is in Class II if it meets both of the following conditions:

(1) It is not in Class VI.

(2) It is never immediately followed by a left parenthesis unless it is immediately preceded by the word FUNCTION in a FUNCTION statement.

10.2 Definition. There are two levels of definition of numeric values, first level definition and second level definition. The concept of definition on the first level applies to array elements and variables; that of second level definition to integer variables only. These concepts are defined in terms of progression of execution; and thus, an executable program, complete and in execution, is assumed in what follows.

There is another variety of definition which refers to when an external procedure may be referenced, and it will be discussed in the next section.

discussed in 7.1.1.3 and 7.1.2.1.2; the second, which refers to when an external procedure may be referenced, will be discussed in the next section.

In what follows, otherwise unqualified use of the terms definition and undefinition (or their alternate forms) as applied to variables and array elements will imply modification by the phrase on the first level.

10.2.1 *Definition of Procedures.* If an executable program contains information describing an external procedure, such an external procedure with the applicable symbolic name is defined for use in that executable program. An external function reference or subroutine reference (as the case may be) to that symbolic name may then appear in the executable program, provided that number of arguments agrees between definition and reference. In addition, for an external function, the type of function must agree between definition and reference. Other restrictions on agreements are contained in 8.3.1, 8.3.2, 8.4.1, 8.4.2, 10.1.3, and 10.1.4.

The basic external functions listed in (8.3.3) are always defined and may be referenced subject to the restrictions alluded to in the foregoing.

A symbolic name in Class III or Class IV is defined for such use.

10.2.2 *Associations That Effect Definition.* Entities may become associated by:

(1) COMMON association.
(2) EQUIVALENCE association.
(3) Argument substitution.

Multiple association to one or more entities can be the result of combinations of the foregoing. Any definition or undefinition of one of a set of associated entities effects the definition or undefinition of each entity of the entire set.

For purposes of definition, in a program unit there is no association between any two entities both of which appear in COMMON statements. Further, there is no other association for common and equivalenced entities other than those stated in 7.2.1.3.1 and 7.2.1.4.

If an actual argument of an external procedure reference is an array name, an array element name, or a variable name, then the discussions in 10.1.3 and 10.2.1 allow an association of dummy arguments with the actual arguments only between the time of execution of the first executable statement of the procedure and the inception of execution of the next encountered RETURN statement of that procedure. Note specifically that this association can be carried through more than one level of external procedure reference.

In what follows, variables or array elements associated by the information in 7.2.1.3.1 and 7.2.1.4 will be equivalent if and only if they are of the same type.

If an entity of a given type becomes defined, then all associated entities of different type become undefined at the same time, while all associated entities of the same type become defined unless otherwise noted.

Association by argument substitution is only valid in the case of identity of type, so the rule in this case is that an entity created by argument substitution is defined at time of entry if and only if the actual argument was defined. If an entity created by argument substitution becomes defined or undefined (while the association exists) during execution of a subprogram, then the corresponding actual entities in all calling program units becomes defined or undefined accordingly.

10.2.3 *Events That Effect Definition.* Variables and array elements become initially defined if and only if their names are associated in a data initialization statement with a constant of the same type as the variable or array in question. Any entity not initially defined is undefined at the time of the first execution of the first executable statement of the main program. Redefinition of a defined entity is always permissible except for certain integer variables (7.1.2.8, 7.1.3.1.1, and 7.2.1.1.2) or certain entities in subprograms (6.4, 8.3.2, and 8.4.2).

In what follows, otherwise unqualified use of the terms definition and undefinition (or their alternate forms) as applied to variables and array elements will imply modification by the phrase "on the first level."

10.2.1 *Definition of Procedures.* If an executable program contains information describing an external procedure, such an external procedure with the applicable symbolic name is defined for use in that executable program. An external function reference or subroutine reference (as the case may be) to that symbolic name may then appear in the executable program, provided that number of arguments agrees between definition and reference. Other restrictions on agreements are contained in 8.3.1, 8.3.2, 8.4.1, 8.4.2, and 10.1.3.

The basic external functions listed in Section 8.3.3 are always defined and may be referenced subject to the restrictions alluded to in the foregoing.

A symbolic name in Class III or Class IV is defined for such use.

10.2.2 *Associations That Effect Definition.* Entities may become associated by:

(1) COMMON association.
(2) EQUIVALENCE association.
(3) Argument substitution.

Multiple association to one or more entities can be the result of combinations of the foregoing. Any definition or undefinition of one of a set of associated entities affects the definition or undefinition of each entity of the entire set.

For purposes of definition, in a program unit there is no association between any two entities both of which appear in COMMON statements. Further, there is no other association for common and equivalenced entities other than those stated in 7.2.1.3.1 and 7.2.1.4.

If an actual argument of an external procedure reference is an array name, an array element name, or a variable name, then the discussions in 10.1.3 and 10.2.1 allow an association of dummy arguments with the actual arguments only between the time of execution of the first executable statement of the procedure and the inception of execution of the next encountered RETURN statement of that procedure. Note specifically that this association can be carried through more than one level of external procedure reference.

In what follows, variables or array elements associated by the information in 7.2.1.3.1 and 7.2.1.4 will be equivalent if and only if they are of the same type.

If an entity of a given type becomes defined, then all associated entities of different type become undefined at the same time, while all associated entities of the same type become defined unless otherwise noted.

Association by argument substitution is only valid in the case of identity of type so the rule in this case is that an entity created by argument substitution is defined at time of entry if and only if the actual argument was defined. If an entity created by argument substitution becomes defined or undefined (while the association exists) during execution of a subprogram, then the corresponding actual entities in all calling program units become defined or undefined accordingly.

10.2.3 *Events That Effect Definition.* Any entity is undefined at the time of the first execution of the first executable statement of the main program. Redefinition of a defined entity is always permissible except for certain integer variables (7.1.2.8 and 7.1.3.1.1) or certain entities in subprograms (6.4, 8.3.2, and 8.4.2).

Variables and array elements become defined or redefined as follows:

(1) Completion of execution of an arithmetic or logical assignment statement causes definition of the entity that precedes the equals.

(2) As execution of an input statement proceeds, each entity, which is assigned a value of its corresponding type from the input medium, is defined at the time of such association. Only at the completion of execution of the statement do associated entities of the same type become defined.

(3) Completion of execution of a DO statement causes definition of the control variable.

(4) Inception of execution of action specified by a DO-implied list causes definition of the control variable.

Variables and array elements become undefined as follows:

(1) At the time a DO is satisfied, the control variable becomes undefined.

(2) Completion of execution of an ASSIGN statement causes undefinition of the integer variable in the statement.

(3) Certain entities in function subprograms (10.2.9) become undefined.

(4) Completion of execution of action specified by a DO-implied list causes undefinition of the control variable.

(5) When an associated entity of different type becomes defined.

(6) When an associated entity of the same type becomes undefined.

10.2.4 *Entities in Blank Common.* Entities in blank common and those entities associated with them may not be initially defined.

Such entities, once defined by any of the rules previously mentioned, remain defined until they become undefined.

10.2.5 *Entities in Labeled Common.* Entities in labeled common or any associates of those entities may be initially defined.

A program unit contains a labeled common block name if the name appears as a block name in the program unit. If a main program or referenced subprogram contains a labeled common block name, any entity in the block (and its associates) once defined remain defined until they become undefined.

It should be noted that redefinition of an initially defined entity will allow later undefinition of that entity. Specifically, if a subprogram contains a labeled common block name that is not contained in any program unit currently referencing the subprogram directly or indirectly, the execution of a RETURN statement in the subprogram causes undefinition of all entities in the block (and their associates) except for initially defined entities that have maintained their initial definitions.

10.2.6 *Entities Not in Common.* An entity not in common except for a dummy argument or the value of a function may be initially defined.

Such entities once defined by any of the rules previously mentioned, remain defined until they become undefined.

If such an entity is in a subprogram, the completion of execution of a RETURN statement in that subprogram causes all such entities and their associates at that time (except for initially defined entities that have not been redefined or become undefined) to become undefined. In this respect, it should be noted that the association between dummy arguments and actual arguments is terminated at the inception of execution of the RETURN statement.

Again, it should be emphasized, the redefinition of an initially defined entity can result in a subsequent undefinition of that entity.

10.2.7 *Basic Block.* In a program unit, a basic block is a group of one or more executable statements defined as follows.

The following statements are block terminal statements:

(1) DO statement.

(2) CALL statement.

Variables and array elements become defined or redefined as follows:

(1) Completion of execution of an arithmetic assignment statement causes definition of the entity which precedes the equals.

(2) As execution of an input statement proceeds, each entity, which is assigned a value of its corresponding type from the input medium, is defined at the time of such association and associated entities become undefined. Only at the completion of execution of the statement do associated entities of the same type become defined.

(3) Completion of execution of a DO statement causes definition of the control variable.

(4) Inception of execution of action specified by a DO-implied list causes definition of the control variable.

Variables and array elements become undefined as follows:

(1) At the time a DO is satisfied, the control variable becomes undefined.

(2) Completion of execution of action specified by a DO-implied list causes undefinition of the control variable.

(3) When an associated entity of different type becomes defined.

(4) When an associated entity of the same type becomes undefined.

10.2.4

10.2.5

10.2.6 *Entities Not in Common.* An entity not in common is initially undefined.

Such entities once defined by any of the rules previously mentioned, remain defined until they become undefined.

If such an entity is in a subprogram, the completion of execution of a RETURN statement in that subprogram causes all such entities and their associates at that time to become undefined. In this respect, it should be noted that the association between dummy arguments and actual arguments is terminated at the inception of execution of the RETURN statement.

10.2.7 *Basic Block.* In a program unit, a basic block is a group of one or more executable statements defined as follows.

The following statements are block terminal statements:

(1) DO statement.

(2) CALL statement.

(3) GO TO statement of all types.

(4) Arithmetic IF statement.

(5) STOP statement.

(6) RETURN statement.

(7) The first executable statement, if it exists, preceding a statement whose label is mentioned in a GO TO or arithmetic IF statement.

(8) An arithmetic statement in which an integer variable precedes the equals.

(9) A READ statement with an integer variable in the list.

(10) A logical IF containing any of the admissible forms given in the foregoing.

The following statements are block initial statements:

(1) The first executable statement of a program unit.

(2) The first executable statement, if it exists, following a block terminal statement.

Every block initial statement defines a basic block. If that initial statement is also a block terminal statement, the basic block consists of that one statement. Otherwise, the basic block consists of the initial statement and all executable statements that follow until a block terminal statement is encountered. The terminal statement is included in the basic block.

10.2.7.1 *Last Executable Statement.* In a program unit the last executable statement (which cannot be part of a logical IF) must be one of the following statements: GO TO statement, arithmetic IF statement, STOP statement, or RETURN statement.

10.2.8 *Second Level Definition.* Integer variables must be defined on the second level when used in subscripts and computed GO TO statements.

Redefinition of an integer entity causes all associated variables to be undefined for use on the second level during this execution of this program unit until the associated integer variable is explicitly redefined.

Except as just noted, an integer variable is defined on the second level upon execution of the initial statement of a basic block only if both of the following conditions apply:

(1) The variable is used in a subscript or in a computed GO TO in the basic block in question.

(2) The variable is defined on the first level at the time of execution of the initial statement in question.

This definition persists until one of the following happens:

(1) Completion of execution of the terminal statement of the basic block in question.

(2) The variable in question becomes undefined or receives a new definition on the first level.

At this time, the variable becomes undefined on the second level.

In addition, the occurrence of an integer variable in the list of an input statement in which that integer variable appears following in a subscript causes that variable to be defined on the second level. This definition persists until one of the following happens:

(1) Completion of execution of the terminal statement of the basic block containing the input statement.

(2) The variable becomes undefined or receives a new definition on the first level.

An integer variable defined as the control variable of a DO-implied list is defined on the second level over the range of that DO-implied list and only over that range.

10.2.9 *Certain Entities in Function Subprograms.* If a function subprogram is referenced more than once with an identical argument list in a single statement, the execution of that subprogram must yield identical results for those cases mentioned, no matter what the order of evaluation of the statement.

If a statement contains a factor that may not be evaluated (6.4), and if this factor contains a function reference, then all entities that might be defined in that reference become undefined at the completion of evaluation of the expression containing the factor.

10.3 DEFINITION REQUIREMENTS FOR USE OF ENTITIES. Any variable referenced in a subscript or a computed GO TO must be

(3) GO TO statement of all types.

(4) Arithmetic IF statement.

(5) STOP statement.

(6) RETURN statement.

(7) The first executable statement, if it exists, preceding a statement whose label is mentioned in a GO TO or arithmetic IF statement.

(8) An arithmetic statement in which an integer variable precedes the equals.

(9) A READ statement with an integer variable in the list.

The following statements are block initial statements:

(1) The first executable statement of a program unit.

(2) The first executable statement, if it exists, following a block terminal statement.

Every block initial statement defines a basic block. If that initial statement is also a block terminal statement, the basic block consists of that one statement. Otherwise, the basic block consists of the initial statement and all executable statements that follow until a block terminal statement is encountered. The terminal statement is included in the basic block.

10.2.7.1 *Last Executable Statement.* In a program unit the last executable statement must be one of the following statements: GO TO statement, arithmetic IF statement, STOP statement, or RETURN statement.

10.2.8 *Second Level Definition.* Integer variables must be defined on the second level when used in subscripts and computed GO TO statements.

Redefinition of an integer entity causes all associated variables to be undefined on the second level during the execution of the program until the associated integer variable is explicitly redefined.

Except as just noted, an integer variable is defined on the second level upon execution of the initial statement of a basic block only if both of the following conditions apply:

(1) The variable is used in a subscript or in a computed GO TO in the basic block in question.

(2) The variable is defined on the first level at the time of execution of the initial statement in question.

This definition persists until one of the following happens:

(1) Completion of execution of the terminal statement of the basic block in question.

(2) The variable in question becomes undefined or receives a new definition on the first level.

At this time, the variable becomes undefined on the second level.

In addition, the occurrence of an integer variable in the list of an input statement in which that integer variable appears following in a subscript causes that variable to be defined on the second level. This definition persists until one of the following happens:

(1) Completion of execution of the terminal statement of the basic block containing the input statement.

(2) The variable becomes undefined or receives a new definition on the first level.

An integer variable defined as the control variable of a DO-implied list is defined on the second level over the range of that DO-implied list and only over that range.

10.2.9 *Certain Entities in Function Subprograms.* If a function subprogram is referenced more than once with an identical argument list in a single statement, the execution of that subprogram must yield identical results for those cases mentioned, no matter what the order of evaluation of the statement.

10.3 DEFINITION REQUIREMENTS FOR USE OF ENTITIES. Any variable referenced in a subscript or a computed GO TO must be defined on the second level at the time of this use.

Any variable, array element, or function referenced as a primary in an expression and any subroutine referenced by a CALL statement must be defined at the time of this use. In the case where an actual argument in the argument list of an external procedure reference is a variable name or an array element name, this in itself is not a requirement that the entity be defined at the time of the procedure reference; however, when such an argument is an external procedure name, it must be defined.

Any variable used as an initial value, terminal value, or incrementation value of a DO statement or a DO-implied list must be defined at the time of this use.

Any variable used to identify an input/output unit must be defined at the time of this use.

At the time of execution of a RETURN statement in a function subprogram, the value (8.3.1) of that function must be defined.

At the time of execution of an output statement, every entity whose value is to be transferred to the output medium must be defined unless the output is under control of a format specification and the corresponding conversion code is A. If the output is under control of a format specification, a correct association of conversion code with type of entity is required unless the conversion code is A. The following are the correct associations: I with integer; D with double precision; E, F, and G with real and complex; and L with logical.

defined on the second level at the time of this use.

Any variable, array element, or function referenced as a primary in an expression and any subroutine referenced by a CALL statement must be defined at the time of this use. In the case where an actual argument in the argument list of an external procedure reference is a variable name or an array element name, this in itself is not a requirement that the entity be defined at the time of the procedure reference.

Any variable used as an initial value, terminal value, or incrementation value of a DO statement or a DO-implied list must be defined at the time of this use.

Any variable used to identify an input/output unit must be defined at the time of this use.

At the time of execution of a RETURN statement in a function subprogram, the value of that function must be defined.

At the time of execution of an output statement, every entity whose value is to be transferred to the output medium must be defined. If the output is under control of a format specification, a correct association of conversion code with type of entity is required. The following are the correct associations: I with integer; and E and F with real.

Editor's Note

Publication of the following Amendment to the proposed American Standard on Specification for General-Purpose Paper Cards for Information Processing, developed by a Subcommittee of ASA Sectional Committee X3, has been authorized by the American Standards Association for the purpose of obtaining comment, criticism and general public reaction, with the understanding that such proposed American Standard has not been finally accepted by ASA as a standard and, therefore, is subject to change, modification, or withdrawal in whole or in part.

On page 286 of the May issue of Communications of the ACM, the proposed American Standard was presented for information and consideration.

Some comment has already been received with regard to the possibility of this being a dual standard, as opposed to preferred and alternate standards.

Some comments have been received about the possibility of dog-earing when a deck of cards contains both types of corner cuts.

Your comments in support, or in opposition, are welcomed and requested. Comments should be addressed to the Secretary X3, Business Equipment Manufacturers Association, 235 East 42 Street, New York, New York 10017.—E.L.

Proposed Amendment to Proposed American Standard on Specification for General-Purpose Paper Cards for Information Processing

1. Paragraph 2.3 of the proposed American Standard Specification for General-Purpose Paper Cards for Information Processing be changed to read as follows:

2.3 *Corners*

2.3.1 Diagonal Corner Cut

2.3.1.1 Dimension. The corner cut shall remove .250 inch ± .016 inch from the long edge and .433 ± .016 inch from the short edge of the card (at a reference angle of 60° to the long edge of the card.)

2.3.1.2 Location

2.3.1.2.1 Preferred Location. The preferred location for the cut shall be at the upper left corner.

2.3.1.2.2 Alternate Location. An alternate location for the cut shall be at the upper right corner.

2.3.2 Other Corners

2.3.2.1 Preferred Corners. All corners, except the diagonally cut corner shall be square. (See 2.1.4 and Figure 1.)

2.3.2.2 Alternate Corners. All corners, except the diagonally cut corner shall be rounded to a nominal radius of .250 inch. The edge of the rounded corner shall fall between two concentric arcs. The center of the arcs is located .242 ± .000 inch from the long edge and .250 ± .000 inch from the short edge of the card. The inner arc shall be 92° and shall have a radius of .242 inch; the outer arc shall have a radius of .272 inch. (See Figure 2.)

2. Add Figure 2.

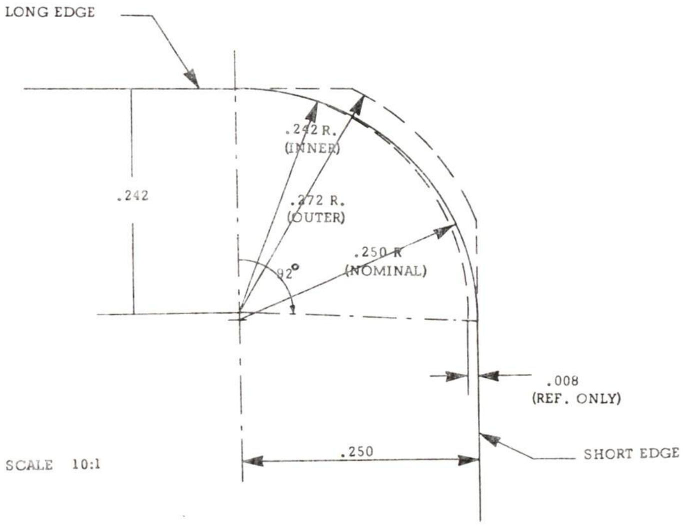


Fig. 2. Rounded corner

Expository Remarks

Several card manufacturers have recently introduced a line of cards having corners rounded to $\frac{1}{4}''$ radius. These cards are designed to improve machine performance by preventing corners from becoming dog-eared and causing feed troubles. It is expected that rounded-corner cards will very quickly come into common usage. Therefore, it is proposed that the proposed American Standard Specification for General-Purpose Paper Cards for Information Processing be amended to include rounded-corner cards.

dec INTEROFFICE MEMORANDUM

DATE November 18, 1966

SUBJECT Increase in Payrolls

TO Ken Olsen

FROM Harry S. Mann

On last week's Works Committee Agenda you had placed an item relating to the kind of marketing and sales people we were looking for and, in effect, why we were looking for so many people in this category. This question has triggered me to make a study of our total payroll situation and I see a very frightening trend which I think should be given prompt attention.

Taking an eight week period in December and January of this year, our hourly payroll averaged $55,200 per week. These people are essentially all production people. The most recent eight week period shows that this payroll has increased to an average of $68,750 per week, or an increase of 24 1/2%. During the same period, the salaried payroll (which includes clerical people, sales people, supervision) increased from $57,850 a week to $82,950 per week. This is an increase of 43 1/2%.

This summary indicates to me that we have been increasing our overhead cost at a rate almost double that of our production workers during the past year.

As you have often stated, the company's goal is to be the lowest cost producer of quality computers in the whole industry. There are many factors that affect cost, of course, including design, efficiency in manufacturing, automation, etc. I submit, however, that we must immediately look at the effect on our cost of this fantastic change in proportion between hourly and salaried people if we are going to achieve our total objective.

This may be a good subject for the next Works Committee Meeting - if not before!

HSM/clw

CC: R. Lassen

Harry

dec INTEROFFICE MEMORANDUM

DATE November 18, 1966

SUBJECT Visit to DEC by Xerox Purchasing & Production Personnel

TO
Ken Olsen
Stan Olsen
Peter Kaufmann
Frank Kalwell
Henry Crouse
Paul McGaunn
Cy Kendrick
Jack Smith
Jim Cudmore
Pat Greene

FROM
Fred Gould FG

Xerox will visit with us on November 30, 1966 for the entire day. Four people are presently scheduled to make the trip, and I have listed their names, titles, below.

They will want to talk on the following subjects:

1. Review of testing procedure by DEC
2. Duplication of DEC test facilities
3. Buying methods, (DEC suggestions solicited)
4. Connector type 144 pcc - Second source, present capacity
5. Xerox requirements for 1967
6. Secrecy Agreement, DEC new products
7. DEC production capacity & reaction time

I will generate an agenda after consulting with the cognizant people as to strategy and convenience to existing schedules.

Mr. James Brown	ISD Purchasing Agent
Mr. Robert Burnham	ISD Electronics Buyer
Mr. Alex Neberenko	Production Control Mgr.
Mr. George Tsilibes	Engineer Liason

/mp

dec INTEROFFICE MEMORANDUM

DATE November 18, 1966

SUBJECT USIA Exhibit

TO Ken Olsen
cc: Ted Johnson
Steve Bowers
Howie Painter

Tim

FROM Tim McInerney

I have arranged for a non-working PDP-8 computer and a quantity of PDP-8 Brochures to be shipped to the designated USIA Warehouse for shipment to the Industrial Design, USA Exhibit in USSR in 1967.

This Exhibit will travel between February and June of 1967 to Kiev, Moscow and Leningrad in the USSR and in September will go to West Berlin, Germany as part of the U. S. Exhibition in the German Industries Fair.

This computer and literature will be shipped from Maynard on Tuesday, November 22.

TJM:jdr

dec INTEROFFICE MEMORANDUM

DATE November 18, 1966

SUBJECT RAND PDP-6

TO Ken Olsen

FROM Jack Shields

I've just read your memo concerning your discussion with Willis Ware at the FJCC. I'll try to briefly update you on this matter.

We are extremely concerned about Rand and the PDP-6 and have no intention of losing interest on this system. Our intentions are clear, as we have recently sent another senior man (Mel Neumann - six years 7090, 3600 experience) to service this account with back up from Bob Brooks. Mel has been instructed to spend full time at Rand for the next few months to insure proper system operation even though our contract doesn't cover this type of service.

I believe that Willis was confused on the documentation problem. Our contract called for weekly reports to Rand on system performance, preventive maintenance performance, etc. We were not submitting these reports. We have, however, always had good documentation on the mod status of the system. I can rationalize why the reports were not filled out, but that is just an excuse. We have taken the following action to rectify this:

1. Mel will spend full time at Rand for the next few months.

2. Copies of all Field Service reports for the system will be filed at Rand.

3. A weekly maintenance schedule will be posted and filled out; copies will be sent to C. Baker.

4. The daily log will be faithfully filled out. Copies will be sent with the other reports on a weekly basis to C. Baker.

5. We will endeavor to convince Rand to purchase a resident engineer contract so we can provide the type of coverage they require.

I'm sure we will have a few more problems at Rand in the future, but I believe we are well on our way toward the establishment of a model installation for both Rand and DEC.

JJS:ned

dec INTEROFFICE MEMORANDUM

DATE November 17, 1966

SUBJECT Advertising

TO Win Hindle
Nick Mazzarese
Stan Olsen

FROM Harry S. Mann

For all products combined we budgeted $56,000 for advertising during the first four months of the year. During this period we spent $112,000 or exactly twice as much as we expected.

The major differences appear in the following products:

Product	Actual	Budget	Overage
8/S	38.4	19.6	18.8
7 & 9	33.4	5.2	28.2
Linc-8	8.3	3.9	4.4

Now that the Product Lines have direct control over advertising, this may be an area you should examine closely.

HSM/clw

CC: Ken Olsen ✓

c - Jim Jordan 11/29

dec **INTEROFFICE MEMORANDUM**

DATE November 17, 1966

SUBJECT Light Pen Memo by Ken Olsen

TO Stu Ogden

FROM Derrick Chin

The switch in our pen opens an aperture so light from the CRT can enter the fiber optic bundles. We have two choices. We can improve the switch as an aperture or we can use the switch electrically to turn on a power supply to activate the photomultiplier: I feel the light pen should be re designed for the following reasons:

1. The user's finger becomestired from the pressure required to keep the switch depressed.

2. The fiber bundle is too thick in diameter limiting the minimum radius of bend before the fibers tend to break.

3. The power supply package is bought outside and is not too reliable especially under high temperatures.

4. The external appearance of the pen is not exactly pleasing to the eye.

Comments:

The thickness of the fiber bundle was chosen in the initial design because of the photomultiplier used. The photomultiplier has nine stages and can be bought from Radio Shack for $11.00. This was the primary reason why it was chosen: availability of replacement and price.

It might be worthwhile to check into the availability of a 10 or more stage photomultiplier at approximately the same price. Then the diameter of the light bundle could be reduced. I might point out that the bundle should have some sort of reinforcement around it for strength and to protect it from user abuse.

A "Y" shaped light bundle has been available as an "off the shelf" for several years. By this means, a light source could be built into the power supply package to provide illumination of the area at which the light bundle is aimed.

I think we can build a power supply cheaper and more reliable than the one we are now buying. Originally the power supply had to be mounted inside the Type 30 Display housing where the space is limited. Now that this line is being discontinued and there is lots of space inside the 300 series Display Units, consideration should be given to buying 1000 to 2000 volt power packs from Wabash Magnetics (somewhere in the midwest). This company already supplies us with the 10,000 volt power pack for the 770 power supply used in all our displays.

The bulb for our light source can be built into this new power supply. The light source should have a spectral response limited to the longer wavelength side of the spectrum so that the photomultiplier will not react to the reflection. The light from the CRT should be filtered at the photomultiplier end so that the photomultiplier reacts only to the blue portion of phosphor output which is of short persistence. These filters are available from Kodak.

Your biggest problems will be to produce a pen which has pleasing appearance and to make a better aperture switch. Why can't we have a pen with smooth curves like a Tektronix probe and with a tapered bushing at the junction of the bundle and pen.

DC/bwf

cc: Ken Olsen

INTEROFFICE MEMORANDUM

DATE November 17, 1966

SUBJECT NOTES PER CONVERSATION WITH AMERICAN CAN COMPANY

TO Al Hanson
Pete Kaufmann
Harry Mann
Ken Olsen

FROM Loren Prentice

The power comes into the building underground. The voltage coming into the building is 1308 volts. There are presently two, 1,000 KVA, dry Westinghouse transformers and one 300 KVA, oil fill, G.E. transformer. The power is transformed from 1308 volts down to 480 volts for their lighting.

The purification and water treatment plant, stainless steel, is going to be transferred, but the pumps and the large Cuno filter to the left of the stairway down into the basement, are going to remain.

The office area on the second floor, approximately 11,000 - 12,000 square feet, is serviced by three air conditioners; one is ten tons and one fifteen tons (new in 1954) and one small office is serviced by one five ton. On the first floor is a five ton air conditioner in the production office area and across the hall, in the area now as you saw the other day where the present cafeteria is, is an office area serviced by a 7-1/2 ton combination air and heat control unit (new in 1956). This area can be serviced either by steam from their boilers or from Maynard Industries. Lighting is fluorscent throughout the plant on 277 volts. The air compressors they plan to leave here are: Two units (new in 1954 which were last serviced in 1960) 500 CFM, 10 x 11, minimum air pressure 125 pounds; One unit, 12 x 13, 400 CFM (new 1958); one 14 x 15, 500 CFM (new in 1960). Distribution of air over the first two floors is ample and uniform over the area.

The floor in the basement area in the center aisle is black top over brick pavement, similar to what we have in building #7. The machine shop area and the rebuilding shop have a skim coat of concrete over brick paving blocks. Approximately 2/3 of the grade floor is concrete. This was laid out for me on a floor plan which I will attempt to put down. The rest of the area, approximately 1/3 to 1/4 of the area, is filled with machine bases which were brought to the same grade level as the concrete. The rest of the floors in this area are in poor condition.

There are two solvent vaults. The one that we saw was originally built for the high voltage accellerator room and has walls approximately 3' thick, but the entrance door to this area is not fireproof and hence has only a "B" and "C" rating. The large vault will house approximately 60, 55 gallon drums. There is a small vault on the first floor, approximately the center of the building, rated for "A", "B" and "C" use and holds approximately 12, 55 gallon drums of solvent.

Fans that will be left in the building are: 12 on the 1st floor, 14,000 cubic feet per minute and; 6 at 2,000 cubic feet per minute on the second floor.

The eight Cyclotherm boilers located in the power house are rated at 30 horse power at 150 pounds maximum. They are using them at 125 pounds. Oil is #2. I can't see any possibility of using these, except perhaps one unit which might be kept for our use to destill our own cleaning solvents. A steam type still for this type of operation, we are now consuming roughly 400 gallons per week, would probably pay for itself.

There is a 1" gas main coming into the building in the basement area on the Walnut Street end of the building. This would be more then ample for any use that we might have for gas heating.

The adjacent parking lot holds 226 cars.

dec INTEROFFICE MEMORANDUM

DATE November 11, 1966

SUBJECT Large Real Time Sharing and Its Relation to DEC's Market.

TO
K. Olsen

FROM
G. Bell

CC: W. Hindle
N. Mazzarese
T. Johnson

Enclosed is a memo which I wrote to outline my thoughts on the form I see the current time sharing systems taking in the next few years.

Since DEC has enough sense to keep mostly at the periphery (the form of the central nervous system is not very well developed yet) these applications areas may be of concern.

Attachment: 1

CCP-181 dmj
10/12/66
Gordon Bell

HIGH DATA RATE TERMINALS TO THE 360/67 FOR USERS WITH REAL TIME REQUIREMENTS, AND DISTRIBUTED COMPUTER NETWORKS

I have a couple of grad students working here.

SUMMARY

Computation carried out throughout an area or community may be considered in terms of a mesh with computing sources, and sinks. When the units or form of what's being distributed is determined, and interconnection problems are solved, "computing energy" may be distributed in a fashion similar to power. Toward this goal, it seems desirable to have closer coupling among our present computers, and other users, who have information in machine readable forms which requires processing. This form of system is shown in Fig. 1.

Aside from the experimental aspects of inter-computer links, and the probable inevitability of such a computing system form, a set of terminal nodes to the present 360/67 computing system at CIT could provide immediate benefits. A central system is favored because of economies of scale, memory cost/bit for large memories, processing speed, file facilities, peripherals, and a wide range of shared common procedures and languages.

A central facility also provides a method of interconnection such that the more sophisticated users involved in language and problem solving procedures design can interact with the naive users who merely want answers. By attempting to provide users with a complete system, the tendency to form secondary computation centers will be discouraged. Thus a lower total overhead is possible since operating system software, languages, peripherals, and files do not have to be duplicated.

The LINC only part of a larger system without the limitations of the LINC in terms of software, memory, etc.

The experimental research process can be significantly changed and made more productive by the use of a single computation system. The total experimental process would consist of experimenter, experiments and single system acting as laboratory assistant, laboratory notebook, and results analyzer. By getting direct evaluation of results from the experiment (in some cases being conducted by the system) the experiment can be carried out more rapidly, without the need to carry data from experiment to computer (or computer to computer) for analysis.

I have a Ph.D. student working on this

OBJECT OF INTERCONNECTIONS

The interconnection is suggested primarily in order that direct benefits can be obtained. As a secondary goal, the author feels these systems must be studied now, as a basis for designing more orderly networks in the future which consist of many processors, and can be separated physically for distance, reliability, and design creation reasons. The modest data rates mentioned are only "what's available now", but developments, for example, in high frequency transmission (200 megabits/sec) on a cable, perhaps, microwave, SSB, etc.) will simplify the (and its costs, services, etc) interconnection.

The present telephone network will probably influence computer form development, more than any other single business or technological development.

next hopefully the structure the specification, at least what its I'M working on!

THE IDEAL SYSTEM

The general computer is shown in Fig. 2. In the ideal system, the transmission delay through elements is short and the links' cost is small. The primary memory is large, homogeneous, and separable into block sizes or parts in terms of its access ports, to match the data rate and data buffer size requirements of the processors to which it connects. Thus, remote independent computers (processor(s) and memory, etc.) are mapped into the central memory, and communication is by memory interaction, with inter-processor and memory communication. Though a computer (processor-memory) may map into central store, some processors may not have access to the complete central memory as a means of forming isolated, protected computers.

In the idealized case, any part of the system (processors, processor and memory, etc.) can be stretched to a different physical location. The criteria for system partitioning is in terms of the least number of interconnections moved, their data rates, etc. The degree of system interaction decreases with the distance from primary memory because data rate is highest there. Therefore, if possible, connections should occur as far from primary memory as possible.

As a computer is removed from the system (by making its memory no longer map into the central memory) the problem of communication between the two computers becomes severe. (In the ideal, any processor can process any part of central memory.) Now, inter-communication must be of the form of deliberate data transfers between the two computers, and the computers can no longer share common monitoring or facilitates.

we're talking to IBM about this

A possible solution is a "slave memory system" in which portions of each memory are considered common parts of central memory, and a remote system would have a "copy" of central memory. Each time a reference either to central memory by a processor or to the remote memory by its processor which changed the common area occurred, a slave action would take place which automatically changed the other area.

The T.Sharing Pledge

In most of these cases mentioned, the cost for forming a peripheral system is higher than the incremental cost if the center could provide the service. Thus, each peripheral system size must be considered carefully in terms of data transmission costs, required reliability at the periphery, and the response time, processing power, and cost at the central facility. Some trivial peripherial systems can actually operate more cheaply than a central facility due to the poor processors/memory match of the large system.

The types of users and community form is given in Fig. 1.

Experimental User: |ADEC Market|

If a memory is being used as a pulse height analyzer for a physics experiment an independent memory is justified. Here, an event consists of a binary number which is used to address one of 4000-8000, 12 to 20 bit memory cells, which is to be incremented at event rates up to 1 megacycle. If a 16 bit tally were sufficient, and data were accumulated in a 64 bit memory word of the memory utilization efficiency would be .25, as opposed to a probable .66 for general processing. As the event data rate approaches .75 micro sec, and since a complete 262,144 byte memory module is used, the processor could be completely inhibited and all output including processing, IO data transmission, files activity, etc. would stop. A high data rate experiment would require a transmission line of about 10-20 megabits per second bandwidth running for 1 to 1000 seconds/experiment. If, however, only the accumulated experimental totals are transmitted, only about 100,000 bits need be transmitted. Assuming minor data editing, a factor of 1000 data reduction (as a minimum) would be expected. It would be desirable to add a small, independent increment of memory for the analyzer data, and yet have it still be addressable as part of the main store.

Tape will be main [illegible]

Data Concentrator: 1 PDP-8 Potential market, but no one to do it !

A similar use of a decentralized computer or buffer would handle the concentration of data sources with buffering, so that data transmission could occur over a single line. Here, typewriter data (100-200 bits/sec) from a large number of users in a single location would be concentrated or collected and sent over a single transmission line at a higher rate (2400 bits/sec). A similar use occurs when process control data is transmitted to a central facility. Here, the local system would concentrate data, and provide local control with capability to administer the process (through perhaps at a reduced capacity) independent of the central facility. The central facility would maintain optimization routines, procedures for the local system, monitor its performance, handle co-ordination with other systems, etc. The ability of a local system might include "dial" for help from another large processing source, as a means of backup.

special, limited computers

Special consoles with peculiar driving characteristics would first be interfaced at the periphery, before integration on a larger scale. These might include teaching machine consoles, special problem oriented consoles, speech I/O devices, etc.

I'm building this now.

Not a DEC mkt. unless Printer etc. special interface no profit

Satellite Batch Processing I/O Equipment:

It may also be desirable to have satellite terminal computers which resemble the present day off line computers, with card reader, line printer, magnetic tape units, plotters, and film IO. These peripheral systems act, to lower the data rates for the central facility. In the case of plotting, local control allows a factor of 1000 bandwidth reduction. They also provide a convenient remote means of handling the data at its source.

emotionally attached to idea

General Displays and Display Processors: 338 + other I/O.

Although general displays can utilize central memory, this form of interconnection is not a general solution for remote displays, unless video is transmitted to satellite displays. In the 360/67 system, fast display(s) operating from central memory may be too costly due to the large module size, and interference with the processors. In general, remote displays must have local buffers, and the only time data will be exchanged with the central system is when computing beyond the capability of the local display-processor is required. The actual performance of the displays and/or displays processor can vary from a local general computer to a buffer with no processing capability. A local system might perform light pen tracking vector drawing, text editing, co-ordinate transformations, and some ability to do general computing. In fact, a display system built around a general purpose computer should be less costly than a specialized "display processor".

low cost PDP-8 or 9 with multiple tubes.

Other Special or General Purpose Systems:

Another important form of intersystem communication is with other central or specialized systems. If the communication is with a general system, procedure sharing will occur and load sharing may be possible. A general system must have the ability to call for service on behalf of its users from a special system (e.g. library).

A library may exist as a dedicated system which is justified by user economies, and any attempt to do generalized problem solving on the library system will only increase the cost of a library transaction. Similarly, an attempt to operate the library as part of the central computing facility may result in increased operating costs job. A library's file access requirements may be realized by files which take advantage of hardware idiosyncrasies, and not tax the file system of a general system which may already be "file access bound". The general system would communicate with libraries around the country, as users request this type of service. Again, instead of directly calling cross country an unsophisticated, non-interactive type user may merely employ a procedure from the central facility which will carry on the necessary dialogue with the appropriate library for the desired information. There would exist a directory of dialogues to carry out with each individualized system, since it will not be possible for two systems to have identical control languages. Though the cost for cross country typewriter communication will inevitably decrease, the ease of having "someone else" get the information, using higher data rate lines would favor this method.

In some instances, a special "National Shrine System" a la SOLOMON, Medlars, etc. may be available to solve, on a batch basis, special problems at a rate 10-1000 faster than the best general hardware.

DECENTRALIZED SYSTEM REQUIREMENTS FROM THE CENTRAL FACILITY

In general, the peripheral processing unit relies on a central facility for all functions which are uneconomical, time consuming or impossible to do at the periphery. The main function required is centralized computation because the peripheral system is weakest in terms of processing capacity, availability of languages, and memory space. The periphery is only acting in a fashion necessary to make the problem digestable (from a scheduling and data rate point of view) to the main or central facility.

Compilations for the peripheral system would be done centrally.

Finally, the periphery would use file storage, printing, plotting, and other functions which are not local. In many cases, it will be necessary for a local system to provide its own buffer tape storage, just because the local needs might saturate a central facility, or become uneconomical or impossible from a data transmission viewpoint.

INTERCONNECTION CONSIDERATIONS

(I've gotten through to IBM on this point, at last, because this is where this system is also weak. The most "elegant" system was the PDP-6 with an ability to access PDP-8's or 680's.

Both the peripheral and central facilities will require hardware and software modules. The interconnection links should be standard serial, synchronous format as terminated by the present standard Dataphones. This rather simple link provides a transmission path independent of distance, which is well defined, universally available, and provides transmission rates of from 1.2 to 40.8 k bits/sec. Hopefully, higher rates will be available in the future. The actual physical lines can be constructed easily or supplied by telephone companies. The degree of isolation is such that a faulty line would not effect either system's reliability.

In addition a software interface at the central facility would allow the link to be assigned to a job. This would be similar to a printer or tape unit assignment, with specification of appropriate data transmission modes.

The scheduling-administrative problem can be simplified by employing a realistic cost function. Thus the user charges are stated in terms of probabilities of expected response time for various computational requests.

For the extremely fast intractive real time process, it may be necessary to allow a user to permanently occupy a section of memory, together with appropriate anomonalies in the scheduler. This design would be necessary, for example, if a central facility were used to fulfill the digital computer role of a hybrid (digital/analog) computation system.

PRESENT SYSTEMS AT CIT WHICH MIGHT INTERFACE WITH THE CENTRAL FACILITY

1. G-21 - Present computation facility
   a. Load sharing
   b. Displays data transmission
   c. Experimental equipment interface (Jesse's MACRO Modules)

2. PDP-8/G-15 Electrical Engineering
   a. Student use
   b. Test device for prototype man-machine interface devices
   c. Interface to processes for study of optimum control system
   d. Interface to FM tape reading system for preliminary analysis of EKG data prior to central pattern recoginition procedures

3. DDP-116 - Psychology - Interface to experiments

4. PDP-5 - Physics - Data converter from photograph scanner to other machine readable forms

5. PDP-7 - Physics (Saxonburg) - direct tie to physics experiments
   a. On line spark chamber
   b. Scanner of bubble chamber data
   c. Pulse height Analyzer

Ray Lindsay should contact: Prof. Tom Strauss, E.E. Dept. as a potential PDP-10 sale.

6. Possible Hybrid analog/digital system

Semi-approved!

The simulations of the system are frightening.... IBM's only sold 100 or so of these at > 2. megabucks/. One ~~sto~~ simulation showed that for 30 users the users got 3%, the system orhd is 97%. I'm learning a bit about simulating eg. DON'T Trust it.

GENERAL PROBLEM OF PROVIDING A "BALANCED SYSTEM"

This is ~~my~~ principle intent now of mine,

To provide an idea of the problem of matching processors to memories, in general, consider the 360/67 at CIT:

Memory data transmission of 2, overlapped .75 microsecond core memory modules:

$$2 \text{ modules} \times \frac{2 \text{ memories}}{\text{modules}} \times \frac{1 \text{ memory word}}{.75 \text{ microsec.}} \times \frac{64 \text{ Bits}}{\text{memory word}} =$$

340. megabits/sec, or the available data rate inherent in the fast core memories, standing alone.

Memory data transmission of Large Core Store

$$4 \text{ modules} \times \frac{64 \text{ bits/module}}{8 \text{ microsec.}} = 32 \text{ megabits/sec.}$$

To the memory system which is capable of supplying or absorbing data at 340 + 32 = 372 megabits/sec. there is connected a memory bus which is used as a transmission path between the arithmetic processor(s) and the Multiplex and Selector Channels (I/O Processors) which absorb or supply data. In the case of the Arithmetic Processor, its data rate is:

(32 bits/instruction + 32 bits/data reference) = 64 bits/instruction and the processing rate is approximately 1.5 microseconds instruction. This gives:

$$\frac{64 \text{ bits}}{\text{instruction}} \times \frac{1 \text{ instruction}}{1.5 \text{ microsecond}} = 42.5 \text{ megabits/sec.}$$

drum

I/O processor data rate is quite small, or for the fastest device, the 2301, the transmission rate is:

$$\frac{1.2 \text{ megabytes}}{\text{sec}} \times \frac{8 \text{ bits}}{\text{byte}} = 9.6 \text{ megabits/sec.}$$

Thus looking at the possible memory cycles available, and those cycles use used:

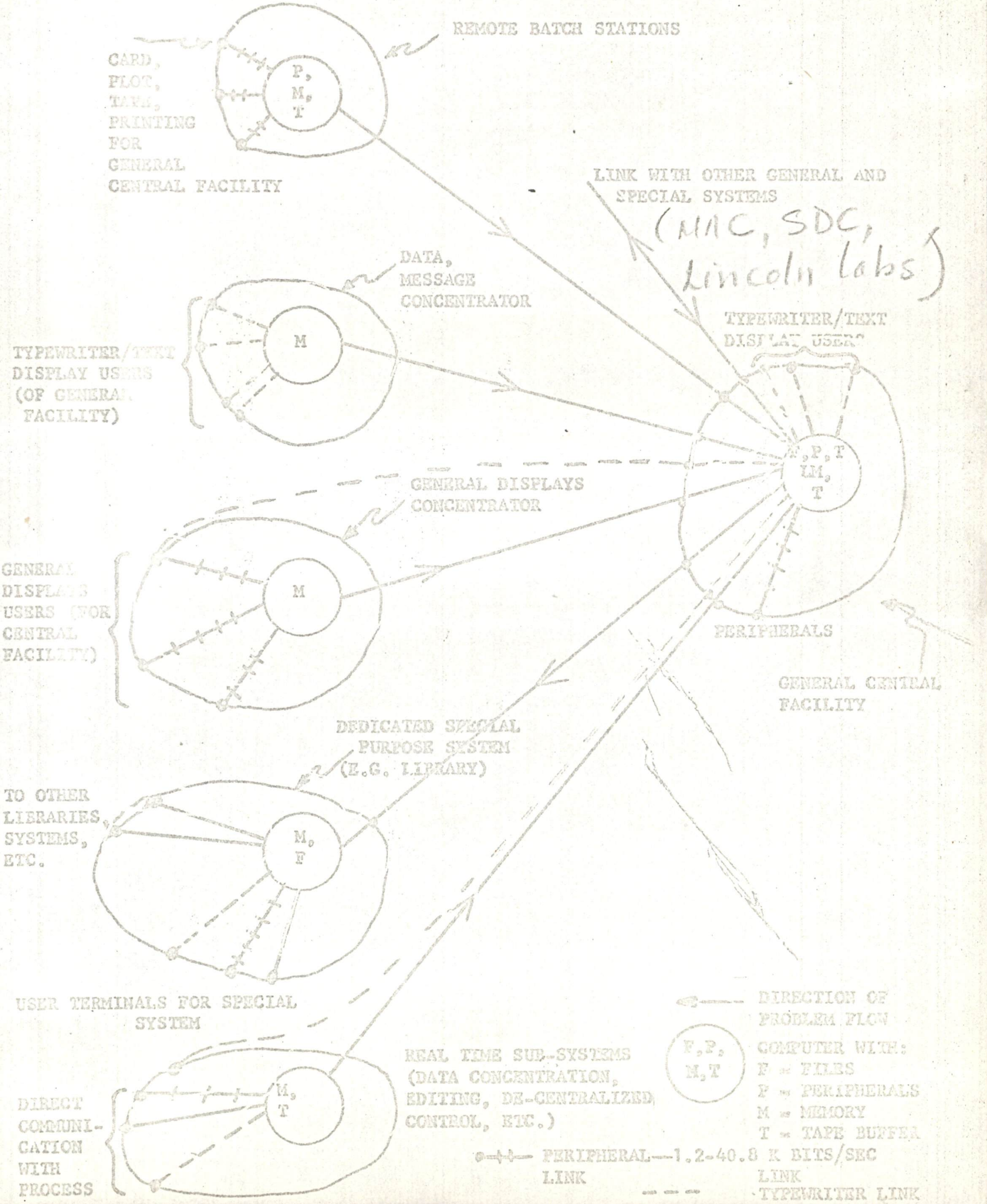
372. megabits/sec available from memory
 50 megabits/sec used by processor + 2301.

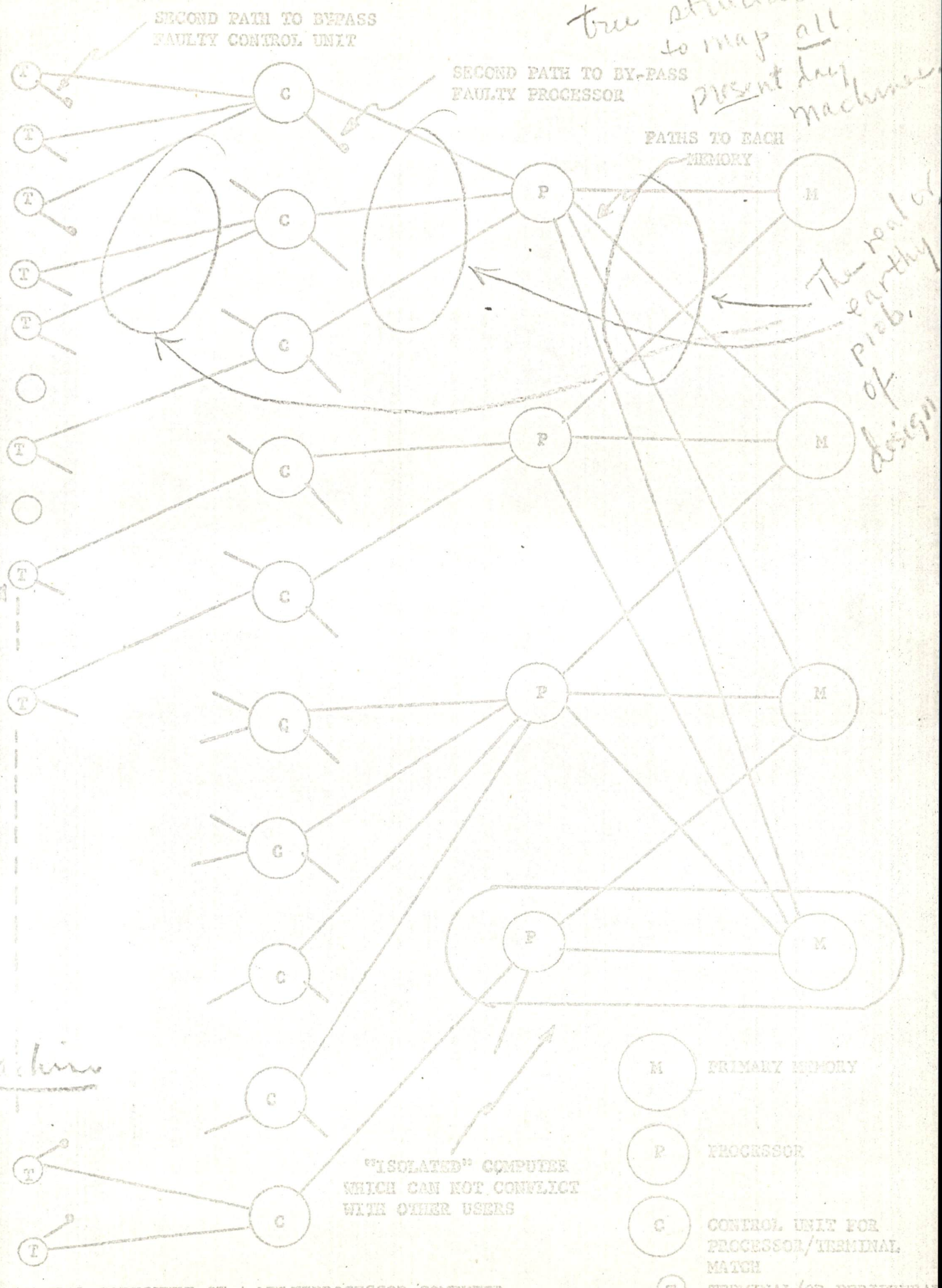
This rather superficial look at the interface between memory and processors (see Fig. 2.) "or I wonder where the bits have gone" is not intended as anything more than a reminder to the author, that it is necessary to determine a performance measurement before embarking on "next generation" synthesis.

Although our present machines may have inefficient inter-connection of components, this may be necessary to achieve the levels of performance.

FORMS OF SYSTEM INTERCOMMUNICATION

Fig. 1.


REMOTE BATCH STATIONS
CARD, PLOT, TAPE, PRINTING FOR GENERAL CENTRAL FACILITY
P, M, T
LINK WITH OTHER GENERAL AND SPECIAL SYSTEMS
(MAC, SDC, Lincoln labs)
DATA, MESSAGE CONCENTRATOR
M
TYPEWRITER/TEXT DISPLAY USERS (OF GENERAL FACILITY)
TYPEWRITER/TEXT DISPLAY USERS
F, P, T LM, T
GENERAL DISPLAYS CONCENTRATOR
GENERAL DISPLAYS USERS (FOR CENTRAL FACILITY)
M
PERIPHERALS
GENERAL CENTRAL FACILITY
DEDICATED SPECIAL PURPOSE SYSTEM (E.G. LIBRARY)
TO OTHER LIBRARIES, SYSTEMS, ETC.
M, F
USER TERMINALS FOR SPECIAL SYSTEM
DIRECT COMMUNICATION WITH PROCESS
M, T
REAL TIME SUB-SYSTEMS (DATA CONCENTRATION, EDITING, DE-CENTRALIZED CONTROL, ETC.)
DIRECTION OF PROBLEM FLOW
F, P, M, T
COMPUTER WITH:
F = FILES
P = PERIPHERALS
M = MEMORY
T = TAPE BUFFER
PERIPHERAL—1.2-40.8 K BITS/SEC LINK
LINK
TYPEWRITER LINK


SECOND PATH TO BYPASS FAULTY CONTROL UNIT
SECOND PATH TO BY-PASS FAULTY PROCESSOR
PATHS TO EACH MEMORY
T
T
T
T
T
T
T
T
T
T
C
C
C
C
C
C
C
C
C
C
P
P
P
P
M
M
M
M
"ISOLATED" COMPUTER WHICH CAN NOT CONFLICT WITH OTHER USERS
M
PRIMARY MEMORY
P
PROCESSOR
C
CONTROL UNIT FOR PROCESSOR/TERMINAL MATCH
T
TERMINAL/OR PERIPHERAL

This rather simple tree structure seems to map all present day machines.

The real ol earthy prob. of design

Direct
eg. TTY, TV

Indirect
Card
Tape

Direct-Machine
A-D.

Fig. 2. GENERAL STRUCTURE OF A MULTIPROCESSOR COMPUTER

dec INTEROFFICE MEMORANDUM

DATE November 9, 1966

SUBJECT Dust Count

TO Ken Olsen
Stan Olsen
Loren Prentice
Harry Mann
Peter Kaufmann
Dick Best

FROM Bob Brown

The representative for Air Controls demonstrated a Bausch & Lomb dust counter system today. The dust level in which we are working is shocking.

The basis of measurement is the number of particles per cubic foot of air taken into the instrument. This air is drawn from the area being monitored.

The results of a few measurements were as follows:

PLACE	X 10^6 Particles/cu. ft.
Photo-Resist Bench	1.2147
2' Above Bench Near Clean Hood	0.968
Top of Lamp Housing in Clean Hood	0.0000
Work Area of Jig	0.0004
In Clean Hood Under Damage Filter	.0200
Almost Anyplace in Diffusion Room	3.0

This information may be interpreted in the following manner:

1) The filter in the clean hood is damaged and needs replacing.

2) The dust count on the photo-resist bench is a large contributor to leak diodes.

3) The dust count in the diffusion room is as bad as our lobby.

The conclusion is that the cleanup and rearrangement of the device area to enforce cleanup is not just necessary, it is imperative.

BB/mf

Ken Olsen

dec INTEROFFICE MEMORANDUM

DATE November 8, 1966

SUBJECT MINUTES OF MEETING ON MACHINE TESTING

TO Attendants of Meeting FROM Jack Shields

A meeting was held Wednesday, 11-2-66, in Ken Olsen's office to talk generally about testing of computers. Those attending the meeting were: Ken Olsen, Saul Dinman, Marv Horovitz, Jack Smith, Bud Dill and Pete Kaufmann.

The meeting started with a bit of testing philosophy that is used in the company on our basic computer products; e.g., Central Processor and Memory. The question of use of high speed readers to speed up the checkout and acceptance test process was discussed and everyone agreed that this would be a good idea. It was agreed, however, that before this step would take place in acceptance testing, a much improved teleprinter test program would be written by Marv and a detailed test procedure for this unit would be submitted by Jack Smith for Field Service approval.

Some discussion took place about the length of time for running tests and it was agreed that the time was based on judgement of people familiar with the units and was adjusted according to the results obtained from field performance.

A discussion of vibration testing took place, however, this one was shelved due to the complexity of description of this test and whether as a useful tool it did more harm than good.

Once again the question was raised as to why Field Service should run test programs at acceptance test time which have previously been run on the computer in the checkout process. The answer was that when acceptance tests became a rubber stamp, Field Service will quickly phase out of the loop.

It was mentioned that the 8/S has a written test procedure which should enable field feed back to close the loop on problems which get by our testing procedure.

It was generally agreed that this was one way problems should be solved rather than trying to re-write the acceptance test to catch the problem at that time. It was pointed out that these procedures were not used for the 7A and 8 although they were written for the 7A at one time.

The concensus of opinion was that these test procedures should be written on the 8 and Jack Smith's group will do this as soon as they can get to it.

The testing of options was brought up during the meeting, but since this is a very involved discussion, it was deferred until it could be discussed at a meeting at a later date.

JJS:ned

dec INTEROFFICE MEMORANDUM

DATE November 7, 1966

SUBJECT Present Status of Foxboro

TO Mike Ford✓
Ted Johnson
Jack Shields
Jack Smith

FROM Ron Wilson

Ellen cc → KHO✓
→ NILK
(over)

As of today all of the problems that we are aware of have been solved or schedules have been established for their solution.

G808E Boards

Twenty have been delivered to Foxboro - Frank Egan will install the boards in one of Foxboro's systems and Foxboro will install the boards in their remaining systems. My point of view is that this is a design improvement and not a mandatory ECO. Outstanding is a requirement for 184 - G808's. Mike Ford has promised these to me in four weeks - I promised to Foxboro within eight weeks.

DM01 - All modules to implement ECO #86 have been received. Frank Egan has them at Foxboro (B141's & S107's).

Mike Ford will deliver ¼ of the modules required to correct our error of shipping Foxboro's DM01's with R line modules instead of S line on 11/7 or 8 - the remaining ½ on 11/14 or 15. These are to be swapped one for one basis.

Ed DeCastro has satisfied Foxboro that the DM01 will operate - He is in the process of issuing an ECO to improve the margins which Frank Egan is implementing in the first one today.

Teletype deliveries we still owe Foxboro Type 35 teletypes. It is probably the most critical item of concern to Foxboro. Foxboro is willing to accept these units without a reader rather than have any more delays.

RWW/jss

Copies to Jim McInerney, Ted, Win, Stan, Nick

11/14/66

INTEROFFICE MEMORANDUM

DATE November 7, 1966

SUBJECT DEC Booth - NEREM 1966

TO K. Olsen

FROM C. Kotsaftis - Cambridge

Although the NEREM booth was attractive and done in good taste, it had several shortcomings.

First of all, the complete lack of a module display was quite a surprise, since NEREM is primarily a components show with the modules attracting the majority of interest. There was sufficient room for a module and logic lab display on the left side of the booth. This area was wasted, since everyone tended to congregate around the two computers.

The second drawback was the physical location of the computers. The intent to emphasize compatibility of the PDP-8 and PDP-8/S by having both teletypes side by side was understandable; however, it did not work. As soon as a demonstration began on one machine, a group of onlookers would crowd around both machines making it difficult to demonstrate the other. If both machines were physically separated, then two demonstrations could have been given simultaneously and more people could have been accommodated. The compatibility of the two machines would have been best demonstrated and explained by the sales people. As it turned out, most people were confused by their proximity.

The most successful part of the show was the handing out of the small Computer Handbook and the Digital Logic Handbook. Having this done by a young lady released the sales people for direct selling. I would suggest the addition of a stockroom man or shipping clerk to physically handle the cartons and keep sufficient quantities of handbooks on hand, since during busy periods this required the full-time attention of one salesman.

In order to avoid future wasting of company resources on trade shows, the following recommendations are suggested:

1. Let the sales office in the trade show area participate in the planning. The local sales personnel are more aware of their immediate market, current trends, and current interests. We also have ultimate responsibility for sales.

2. Have a pre-show meeting of all those on booth duty. Typical topics for discussion should be: The new products on display, if any, that we have not been informed about. (This happened at the Spring Joint Conference). Familiarization with what is being

demonstrated; i.e., how the demonstration works. An explanation of the "theme" if there is one or of the message we are trying to get across.

3. A post show meeting to discuss the results of the show in terms of reactions to booths, displays, demonstrations, etc. Or, in short, did we achieve the results we wanted?

The question boils down to whether or not trade shows are a significant part of our total sales effort. If not, then possibly we should stop participating in shows as some of our competition has. If on the other hand we intend to continue using trade shows, we should provide the same planning, support, and follow-up which would be required for any other sales or marketing programs.

My final comment is in regards to the general lack of interest and support from Maynard. It is appropriate to note that many Maynard personnel who could not find time to support the NEREM show had no difficulty scheduling a week for a West Coast show.

P. MAXCY
11[illegible]66

STORAGE AREA	LINE	PERSON RESPONSIBLE	BLDG. NO.	KEY NO.	TYPE OF ARTICLES STORED
1	Small Comp. Prod.	J. Smith	11	5	teletype parts-logic parts-packing foam-obsolete mechnical parts
2	Small Comp. Prod. Line	N. Mazzarese	11	38 W	
3	Module Production	C. Kendrick	11	5	obsolete production machinery
4	Personnel	R. Lassen	11	4	Company entertainment equipment
5	Direct Mail	T. McInerney	8	27	Promotional literature
6	Print Ship	D. Lewis	11	6	misc. sales literature
7	Programing	H. Shepard	11	16	packing materials
8	Programing	H. Shepard	11	16	binders
9	Advertising	D. Ward	11	16	standard maintenance manuals
10	Purchasing	H. Crouse	11	11	graphic supplies-paper supplies-wire supplies
11	Administration	K. Olsen	11	40	old Company history
12	Plant Engineer	A. Hansen	11		maintenance supplies-pipes and plumbing equipment.
13	Plant Engineer	A. Hansen	11		
14	Accounting	R. Dill	8 A	4	financial records-registers-pay roll records
15	Accounting	R. Dill	8 A	4 W	IBM tabulating equipment
16	Large Comp. Prod.	S. Mikulski	11	32	manuals-papers-old records
17	Trade Shows	T. McInerney	11	27	Trade Show Material
18	Traffic	F. Kalwell	8	8 W	old F.G.'s records- supplies
19	Power Supply Prod.	R. Maxcy	8	3 W	raw materials
20	Purchasing	H. Crouse	8 A	11	graphic supplies-paper supplies-wire supplies
21	Small Comp. Prod.	J. Smith	8 A	5	teletype parts-logic parts-packing foam

dec INTEROFFICE MEMORANDUM

DATE November 4, 1966

SUBJECT Employee Entrance/Exit -- Building 11

TO K.H. Olsen

FROM Bob Lassen

cc: Peter Kaufmann
Loren Prentice
Cy Kendrick

- - -

We have agreed with Loren Prentice to allow the people in the silk screening area to enter and exit through the rear door in Building 11. This will not interfere with the company's security regulations as the rear door must remain unlocked because it is used by other companies in the mill. We have also placed a time clock near the rear door for our employees' convenience. In addition we will install two or three outside floodlights at the rear of Building 11 for the safety of women employees entering or leaving during periods of darkness.

Students and instructors in the Training Department will continue to use the front door of Building 11 as they do not have access to other parts of the plant.

/bjz

INTEROFFICE MEMORANDUM

DATE November 4, 1966

SUBJECT Inhouse Integrated Circuits

TO Ken Olsen W. Hindle
Dick Best P. Kaufmann
Stan Olsen N. Mazzarese

FROM Bob Brown

WHAT

Integrated circuits can be developed in several ways. For instance, isolation may be accomplished by any one or more of the following methods: (a) p-n junction (b) ceramic and glass dielectric or (c) mesa as well as beam leads are all isolation modes.

Just as there are a multitude of isolation modes there are also many circuits in which the isolation may be made by using clusters of common mode diodes and transistors set down on hybrid circuit mounting strates.

If we follow logical step by step development program we shall progress as follows:

> First it is necessary to develop a production staff which can, in a routine manner, handle standard diffusion, cleaning, metalizing and masking operations.
>
> For this to function well and reliable, it must be routine as in production of a couple of diodes and/or a transistor.
>
> The second step is to be sure to have at hand the techniques for producing the devices which are to be incorporated in the IC's.
>
> One should as a third step make circuits which do not require isolation modes such as enumerated above.
>
> Step four is to develop the simplest isolation mode which should also be as free as possible of parasitics.
>
> Fifth is the production of circuits which have large usage.

I feel that in order to make the most versatile circuits we should, where practical, integrate active elements and as much circuit as possible and hybridize them with screened (trimmed) resistors and capacitors.

By taking a step by step approach to the problem we can make each step pay its way.

THE HOW OF IT

(1) It is essential that the basic technology which is envolved in step one be firmly established. Problems experienced at this stage which are not solved will be around to cause trouble in all future stages of development. It requires finesse to produce semiconductor devices. It has never been done successfully with brute force.

(2) When we have developed a transistor we shall have accomplished step two.

(3) Recent discussion with Saul Dinman, Russ Doane and Tom Stockebrand has indicated that there may be common mode circuits which we can produce at a real value to the Company, and accomplish step three.

(4) Of the several isolation modes, the simplist technique for isolating devices in an integrated circuit was described by Schroder of Fairchild at the IEEE Electron Devices meeting. They call it mesa isolation. This system will require development of far fewer techniques than any of the other modes and appears to result in far better isolation.

(5) Sometime prior to this phase(step five), we should obtain mask making equipment and develop a masking capability. The major effort during step five will be in quick change of masks to deliver to the circuit people circuits; experimental, and finalized. We should have developed multiple layer circuit techniques during this step.

THE WHY OF IT

While at the conference in Washington (Electron Devices Conference) I talked to numerous old friends.

Many of them are setting up inhouse device capabilities around the nation. In our discussions we always seemed to get around to the question, "Why produce inhouse devices and Ic's?". The universal answer was that there was no real basis for competing if everyone used only those IC's which were offered on the open market.

The common opinion was that a semiconductor group working with circuit designers within the company could produce circuits with which the semiconductor companies could not compete. This would also give the company the opportunity to use those capabilities which made them successful in the past.

There was much talk of L.S.I. (Large Scale Integration) at the conference. Most of the problems involved are of the sort best solved by a company such as DEC.

BB/mf

dec INTEROFFICE MEMORANDUM

DATE November 4, 1966

SUBJECT ANNUAL PHYSICAL EXAMINATIONS--KEY PEOPLE

TO Ken Olsen

FROM Bob Lassen

Arrangements for annual physical examinations are complete. Dr. Elmer Purcell of Concord and Dr. Houck have agreed to a rather extensive examination designed to fit the medical history and age of each of each employee. Dr. Purcell has an excellent reputation and is enthusiastic about handling this for us.

The cost of the physicals will be about $50 per person, and the results will be forwarded to Dr. Houck. Elsa will handle the appointments and will remind people when they are to take their examination.

All people concerned have been notified.

RTL/jfr

digital EQUIPMENT CORPORATION
MAYNARD, MASSACHUSETTS

Elsa Carlson — 11/4/66

I agree with Pat Greene that we should not contribute to this Conference. I think Pat may already have talked with the man when he called. You can check Pat on this as he is out this afternoon.

Ken

dec INTEROFFICE MEMORANDUM

DATE Nov. 3, 1966

SUBJECT

TO Win Hindle

FROM Jack MacKeen

Win, Pat and I both feel that the only benefit received from this conference is product exposure and that by exhibiting, we are supporting it as far as we wish to. We do not recommend any further support.

1967 International Congress on Magnetism

Boston, Massachusetts **10-16 September 1967**

sponsored by **International Union of Pure and Applied Physics** • **American Institute of Physics**

in cooperation with **Conference on Magnetism and Magnetic Materials**

Address reply to:
Sperry Rand Research Center
100 North Road
Sudbury, Massachusetts 01776

October 17, 1966

Mr. Kenneth H. Olsen
Digital Equipment Corp.
146 Main Street
Maynard, Massachusetts

Dear Mr. Olsen:

The 1967 International Congress on Magnetism is a major scientific event which will take place next September in conjunction with the annual (U.S.) Conference on Magnetism and Magnetic Materials. This will be the first time for the international meeting to be held in the United States, and the New England area has been selected as the site because of the many educational and industrial organizations here which have active interests in the field of magnetism. Previous International Congresses were held in Nottingham (England, 1964), in Kyoto (Japan, 1961), and in Grenoble (France, 1958).

Everything is being done to make the Congress an outstanding one. Approximately 50 invited papers and 300 contributed papers will be presented to about 1200 scientists and engineers coming from all parts of the United States and abroad. The outstanding scientific events should be matched with equally attractive social events. This is a unique opportunity to show our visitors the beauty of the area, give them an opportunity to engage in informal, relaxed discussions with members of the local scientific and engineering community, and demonstrate that New England hospitality is second to none.

In order to do this we need the help of the research laboratories and industrial organizations located nearby. They will benefit from having this important Congress in Boston by being able to send their scientists and engineers to the technical sessions at low cost, but at the same time are collectively in the position of hosts for the participants coming from distant places. We, therefore, are asking for financial help to provide the amenities which will mean so much to the success of the Congress. Contributions will be acknowledged in the printed program if the donor so desires.

I will call you in a week or two, and will be pleased to try to answer any questions you may have.

Sincerely yours,

Richard W Damon

R. W. Damon
Local Committeeman

RWD/d

dec INTEROFFICE MEMORANDUM

DATE November 3, 1966

SUBJECT PLANT TOURS

TO Bob Lassen ✓ Ken Olsen
Win Hindle Ted Johnson
Mike Ford Ken Gold
Bob Lane

FROM Al Hanson

Please notify the Security Department (Al Hanson - Ext. 379 or Judy French - Ext. 421) with the following information before a plant tour is made: (A plant tour being described as 6 or more persons touring the plant.)

The company or companies involved or groups involved. (Boy Scouts, Girl Scouts, etc.)

The time the tour is to take place.

Those buildings that are to be covered by the tour.

The starting and ending places of the tour.

Those persons conducting the tour should:

Instruct the people with them not to straggle off in all directions, but to stay with the group.

In general, tours should be conducted with enough guides to provide one guide for every 5 or 6 persons.

These guides should be familiar with the route that they are going to take; toilet facilities, areas under which we maintain security and they should not enter, etc.

As the buildings are almost always under construction and we are moving heavy equipment in and out, your cooperation is solicited to provide for the safety of the people you are conducting through the plant, respect the security we wish to maintain and to save embarrassment both to ourselves and our guests.

1.11.66

MSG NO 33
TO ELSA CARLSON; CC NICK MAZZARESE
FROM GEOFF FINCH

REF MAURICE WILKES, CAMBRIDGE CS/66/11

I AM VISITING HIM TUESDAY 1ST NOV. WITH ESSENTIALLY NO GOOD NEWS FROM THE PLANT. PLEASE PUT TOP PRIORITY ON GETTING THE EAE DELIVERED THIS MONTH WITH NECESSARY SPARES AND INSTALLATION KIT AS RECOMMENDED BY LARRY SLIGMAN FOR EARLY PDP-7 AT MATHS LAB.
I HAVE NOT SEEN M.W'S LETTER TO KEN.

ALS.

digital EQUIPMENT CORPORATION
MAYNARD, MASSACHUSETTS

SALES CALL REPORT NO. 12696

DATE 11/1/66

FIRM Electronic Data Systems (Pepsi)

STREET 1500 Peach Tree Center
230 Peach Tree St. N.W.

CITY Atlanta, Georgia

SALESMAN Ron Bassin

OFFICE AREA New York

AREA CODE 404 PHONE NO. 577-3374

PHONE (OURS THEIRS) □ LETTER □ VISIT □

PERSONS CONTACTED EXTENSION

(D) Jack Archer

EST. ANNUAL POT.	CK.		CK.	PRODUCT	CK.	TYPE
		NEW		MODULES		
UNDER $20K		OLD	X	A/D		
$20 - 50K		HIGH		COMPUTORS	X	
$50 - 150K		MED.		SPECIAL SYSTEMS		
$150K UP	X	LOW		OTHER		

REMARKS

1. EVALUATION We lost the sale of 25 PDP-8/S's to "CCC" DDP 116 for the following reasons quote:
   1. "CCC" is marketing orientated DEC is not.
   2. "CCC" will lease with an option to buy!!!

Please note the difference in cost between the PDP-8/S and the DDP116. Jack Archer agreed that the 8/S will do the job. EDS will do the programming .

ACTION TO BE TAKEN

✓

K. Olsen, S. Olsen, M. Ford, T. Johnson FOLLOW-UP DATE BY

SPECIAL COPIES TO

HOME OFFICE COPY