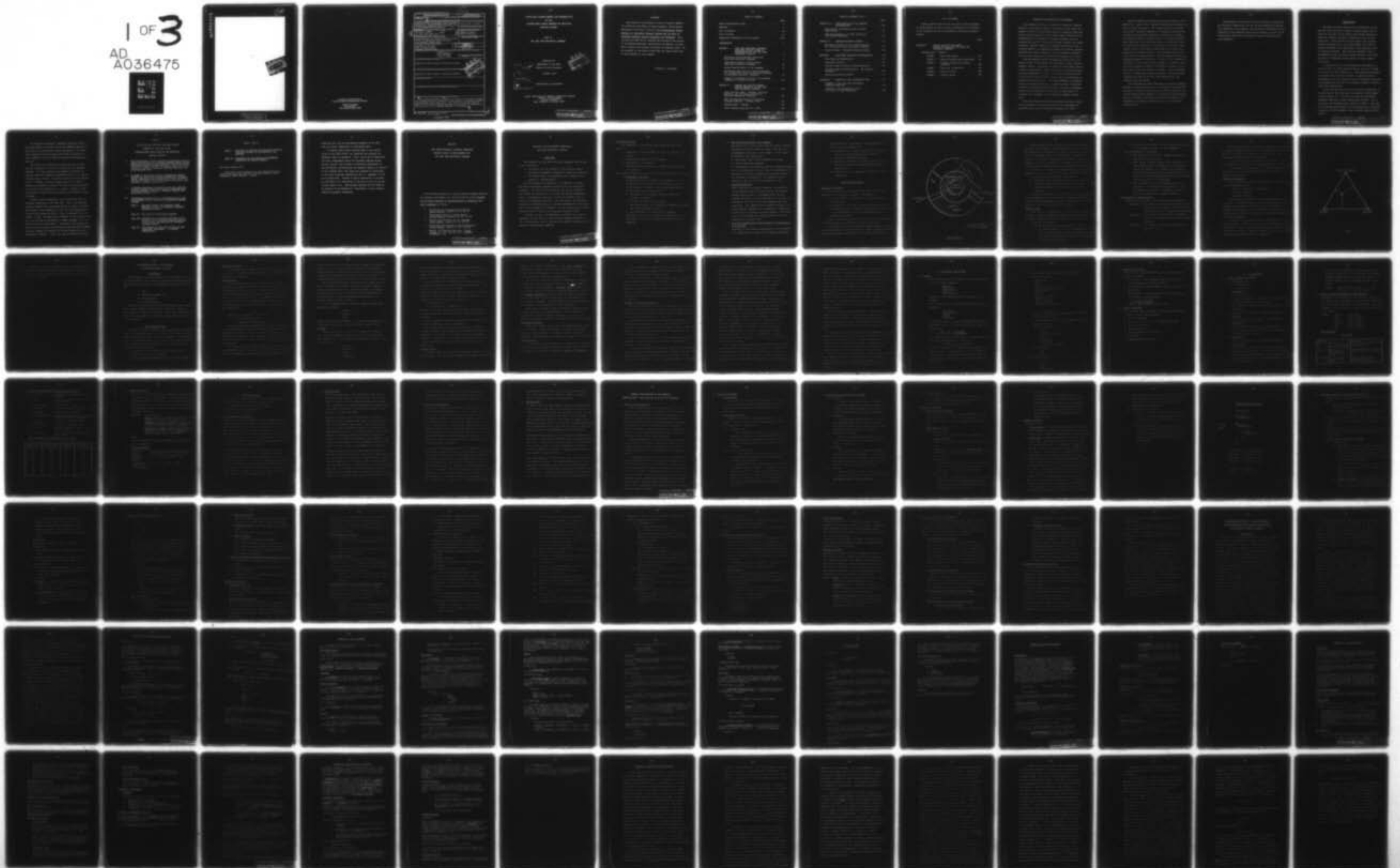
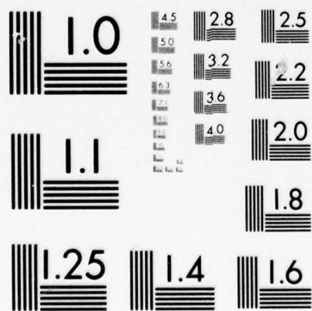

AD-A036 475
PURDUE UNIV LAFAYETTE IND PURDUE LAB FOR APPLIED IND--ETC F/G 9/2
SIGNIFICANT ACCOMPLISHMENTS AND DOCUMENTATION OF THE INTERNATIO--ETC(U)
JAN 77
N00014-76-C-0732
UNCLASSIFIED
NL
1 OF 3
AD
A036475

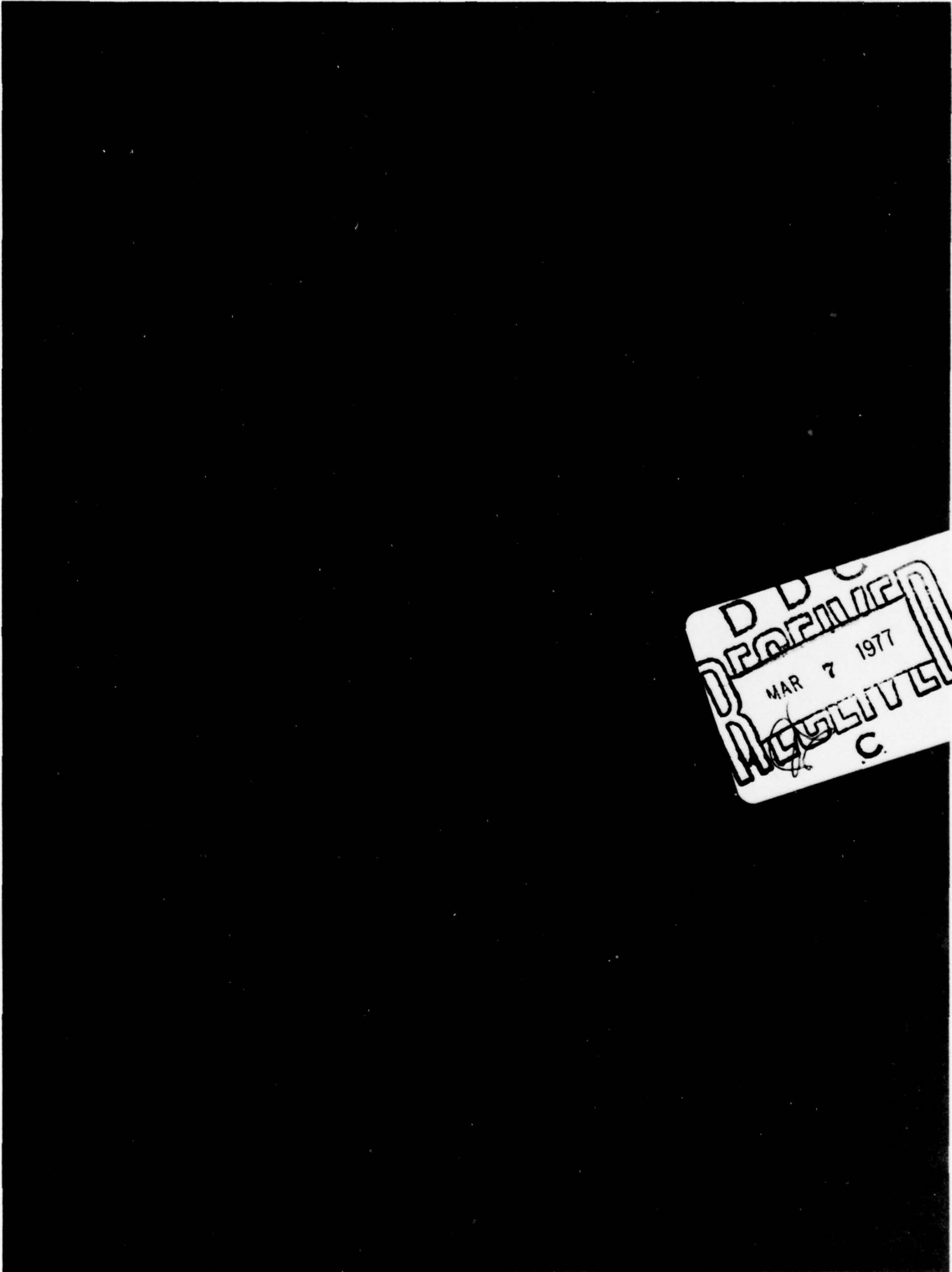
03647



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

ADA036475

(12)



DISTRIBUTION STATEMENT A
Approved for public release,
Distribution Unlimited

This Report is Published as Part of
Engineering Experiment Station Bulletin 143 Series

Schools of Engineering
Purdue University
West Lafayette, Indiana 47907

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ~~NR049-388~~ (6)	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SIGNIFICANT ACCOMPLISHMENTS AND DOCUMENTATION OF THE INTERNATIONAL PURDUE WORKSHOP ON INDUSTRIAL COMPUTER SYSTEMS. PART II. THE LONG TERM PROCEDURAL LANGUAGE.		5. TYPE OF REPORT & PERIOD COVERED Final 2/1/76 - 1/31/77
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (9) Final rept. 1 Feb 76 - 31 Jan 77.		8. CONTRACT OR GRANT NUMBER(s) (15) N00014-76-C-0732
9. PERFORMING ORGANIZATION NAME AND ADDRESS Purdue Laboratory for Applied Industrial Control, Schools of Engineering, Purdue University, West Lafayette, Indiana 47907		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Department of the Navy, Office of Naval Research, Arlington, Virginia 22217	(11)	12. REPORT DATE January 1977
		13. NUMBER OF PAGES 295 + xiv
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) (12) 307p.		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

DDC RECEIVED MAR 7 1977 C

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Approved for public release; distribution unlimited.

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This volume represents Part 2 of a six volume set reproducing the major work accomplished by the International Purdue Workshop on Industrial Computer Systems during the past eight years. This material is reprinted from the Minutes of the several individual meetings of the Workshop and represents the work carried out by the standing committees of the Workshop.

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-014-6601

408244

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

SIGNIFICANT ACCOMPLISHMENTS AND DOCUMENTATION OF THE INTERNATIONAL PURDUE WORKSHOP ON INDUSTRIAL COMPUTER SYSTEMS

PART II
THE LONG TERM PROCEDURAL LANGUAGE

Prepared for
Department of the Navy
Office of Naval Research

January 1977

D D C
MAR 7 1977
ACCESSION for
NTIS White Section
DDC Buff Section
UNANNOUNCED
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
DIST. AVAIL. and/or SPECIAL
A

Distribution is Unlimited

Purdue Laboratory for Applied Industrial Control
Schools of Engineering
Purdue University
West Lafayette, Indiana 47907

PRECEDING PAGE BLANK-NOT FILMED

FOREWORD

This material is published as part of Contract N00014-76-C-0732 with the Office of Naval Research, United States Department of the Navy, entitled, The International Purdue Workshop on Industrial Computer Systems and Its Work in Promoting Computer Control Guidelines and Standards. This contract provides for an indexing and editing of the results of the Workshop Meetings, particularly the Minutes, to make their contents more readily available to potential users. We are grateful to the United States Navy for their great help to this Workshop in this regard.

Theodore J. Williams

PRECEDING PAGE BLANK-NOT FILMED

TABLE OF CONTENTS

Page

Report Documentation Page i

Foreword v

List of Figures ix

List of Tables xi

Background Information on the Workshop xiii

INTRODUCTION 1

SECTION I - LONG TERM PROCEDURAL LANGUAGES COMMITTEE WORKING PAPERS ON REQUIREMENTS FOR THE LONG TERM PROCEDURAL LANGUAGE 5

Objectives and Environment Definition Long Term Procedural Language 7

Supplemental Report on Requirements Long Term Procedural Language 17

General Specification of the Language 37

Preliminary Description of Chosen Functions For Incorporation Into Proposed Specification of Long Term Procedural Language 57

Summary of Preferable Features of Procedural Language for Industrial Usage 99

SECTION II - WORKING AND POSITION PAPERS - TASKING PROPOSALS FOR THE LONG TERM PROCEDURAL LANGUAGE 105

LTPLC Position Paper - Tasking, Interrupt Handling, and the Synchronization of Parallel Computation Processes 107

Task Interactions Long Term Procedural Language Committee - Working Paper 115

Working Paper - Tasking 127

Joint Tasking Proposals for a LTPL 167

PRECEDING PAGE BLANK-NOT FILMED

TABLE OF CONTENTS (Cont.)

Page

SECTION III - INPUT/OUTPUT AND FILE HANDLING CONSIDERATIONS 189

Input/Output Including Process Oriented Input/Output 191

Some Requirements to an LTPL Concerning I/O and File Handling 207

SECTION IV - HARDWARE/SOFTWARE LINKAGES 211

The Pearl Proposal for the System Division of a Process Oriented Programming Language 215

Position Paper - Hardware-Software Linkage 233

SECTION V - SOME OTHER PERTINENT CONSIDERATIONS . . . 237

Data Types and Manipulation. 239

Acceptance Criteria. 243

Position Paper on Static Program Structure 249

Position paper on Extensibility: The Operator Statement 259

Operating Systems Interface 271

SECTION VI - REVIEWS OF LTPL DEVELOPMENT WORK 277

Comments to the Long Term Procedural Language Committee 279

Comments on the Assessment of PL/1 As a Basis of LTPL Development 285

LIST OF FIGURES

Figure numbers used here are the same as those assigned to these figures in their previous publication in the Minutes of the International Purdue Workshop on Industrial Computer Systems.

Page

SECTION II - WORKING AND POSITION PAPERS - TASKING PROPOSALS FOR THE LONG TERM PROCEDURAL LANGUAGE

Working Paper - Tasking

FIGURE 1 - Real-Time Process 155

FIGURE 2 - Typical Message Input Processing . 155

FIGURE 3 - Possible Interaction Between Processes 156

FIGURE 4 - Data Base I/O Routines 157

FIGURE 5 - Message Systems 158

FIGURE 6 - Message Queues 159

BACKGROUND INFORMATION ON THE WORKSHOP

The International Purdue Workshop on Industrial Computer Systems, in its present format, came about as the result of a merger in 1973 of the Instrument Society of America (ISA) Computer Control Workshop with the former Purdue Workshop on the Standardization of Industrial Computer Languages, also co-sponsored by the ISA. This merger brought together the former workshops' separate emphases on hardware and software into a stronger emphasis on engineering methods for computer projects. Applications interest remains in the use of digital computers to aid in the operation of industrial processes of all types.

The ISA Computer Control Workshop had itself been a renaming in 1967 of the former Users Workshop on Direct Digital Computer Control, established in 1963 under Instrument Society of America sponsorship. This Workshop in its annual meetings had been responsible for much of the early coordination work in the field of direct digital control and its application to industrial process control. The Purdue Workshop on Standardization of Industrial Computer Languages had been established in 1969 on a semiannual meeting basis to satisfy a widespread desire and need expressed at that time for development of standards for languages in the industrial computer control area.

The new combined international workshop provides a forum for the exchange of experiences and for the development of guidelines and proposed standards throughout the world.

PRECEDING PAGE BLANK-NOT FILMED

Regional meetings are held each spring in Europe, North America and Japan, with a combined international meeting each fall at Purdue University. The regional groups are divided into several technical committees to assemble implementation guidelines and standards proposals on specialized hardware and software topics of common interest. Attendees represent many industries, both users and vendors of industrial computer systems and components, universities and research institutions, with a wide range of experience in the industrial application of digital systems. Each workshop meeting features tutorial presentations on systems engineering topics by recognized leaders in the field. Results of the workshop are published in the Minutes of each meeting, in technical papers and trade magazine articles by workshop participants, or as more formal books and proposed standards. Formal standardization is accomplished through recognized standards-issuing organizations such as the ISA, trade associations, and national standards bodies.

The International Purdue Workshop on Industrial Computer Systems is jointly sponsored by the Automatic Control Systems Division the Chemical and Petroleum Industries Division, and the Data Handling and Computations Division of the Instrument Society of America, and by the International Federation for Information Processing as Working Group 5.4 of Technical Committee TC-5.

The Workshop is affiliated with the Institute of Electrical and Electronic Engineering through the Data Acquisition and Control Committee of the Computer Society and the Industrial Control Committee of the Industrial Applications Society, as well as the International Federation of Automatic Control through its Computer Committee.

INTRODUCTION

The Office of Naval Research of the Department of the Navy has made possible an extensive report summary and indexing of the work of the International Purdue Workshop on Industrial Computer Systems as carried out over the past eight years. This work has involved twenty-five separate workshop meetings plus a very large number (over 100) of separate meetings of the committees of the workshop and of its regional branches. This work has produced a mass of documentation which has been severely edited for the original minutes themselves and then again for these summary collections.

A listing of all of the documentation developed as a result of the U.S. Navy sponsored project is given in Table I at the end of this Introduction. The workshop participants are hopeful that it will be helpful to others as well as themselves in the very important work of developing guidelines and standards for the field of industrial computer systems in their many applications.

By far the most prolific of the technical committees of the workshop in the production of proposals, working papers, position papers, etc., has been the Long Term Procedural Languages Committee, usually abbreviated as LTPL-C, and in particular its European branch committee, LTPL-E. This part of the documentation series for the Workshop will be devoted completely to LTPL documents, most of which originated with LTPL-E.

PRECEDING PAGE BLANK-NOT FILMED

The Long Term Procedural Languages Committee (LTPL-C) was established at the same time as the FORTRAN Committee to investigate the possibility of the development of an industrial computer systems language which might eventually replace FORTRAN as the most important procedural language for such use.

The work of the committee was early directed to look into the possibility of PL/1 as the basis of the candidate language. For this purpose an arrangement was worked out with the ANSI X3J1 Committee responsible for PL/1 to organize a Working Group, X3J1.4, to develop a process control version of PL/1. However, this work lagged because of a lack of sufficient personnel to maintain the pace of work established by the parent X3J1 Committee. As a result, interest in this area lagged in America and was picked up by the LTPL-E group in Europe.

Several recent developments (as of 1976) have lent considerable importance to the LTPL work. First, the European Economic Community has encouraged LTPL-E to submit a proposal for funding by the Community to carry out the language specification work planned by LTPL-E. This is expected to be approved. Second, the Department of Defense of the United States is in the process of developing a "common language" for U.S. service use to be called DOD-1. The several LTPL groups have been actively reviewing the preliminary specifications (TINMAN) for this language as developed by the HOL Committee of the Department of Defense. Third, the work of ISO/TC 97/SC 5/WG 1

TABLE I

A LIST OF ALL DOCUMENTS PRODUCED IN THIS

SUMMARY OF THE WORK OF THE

INTERNATIONAL PURDUE WORKSHOP ON INDUSTRIAL

COMPUTER SYSTEMS

1. The International Purdue Workshop on Industrial Computer System and Its Work in Promoting Computer Control Guidelines and Standards, Report Number 77, Purdue Laboratory for Applied Industrial Control, Purdue University, West Lafayette, Indiana, Originally Published May 1976, Revised November 1976.

2. An Index to the Minutes of the International Purdue Workshop on Industrial Computer Systems and Its Predecessor Workshops, Report Number 88, Purdue Laboratory for Applied Industrial Control, Purdue University, West Lafayette, Indiana, January 1977.

3. A Language Comparison Developed by the Long Term Procedural Languages Committee - Europe, Committee TC-3 of Purdue Europe, Originally Published January 1976, Republished October 1976.

4-9. Significant Accomplishments and Documentation of the International Purdue Workshop on Industrual Computer Systems.

Part I - Extended FORTRAN for Industrial Real-Time Applications and Studies in Problem Oriented Languages.

Part II - The Long Term Procedural Language.

Part III - Developments in Interfaces and Data Transmission, in Man-Machine Communications and in the Safety and Security of Industrial Computer Systems.

Part IV - Some Reports on the State of the Art and Functional Requirements for Future Applications.

TABLE I (Cont.)

Part V - Documents on Existing and Presently Proposed Languages Related to the Studies of the Workshop.

Part VI - Guidelines for the Design of Man/Machine Interfaces for Process Control

All dated January 1977.

The latter seven documents are also published by the Purdue Laboratory for Applied Industrial Control, Purdue University, West Lafayette, Indiana.

Committee will also be considering languages of the LTPL type and active cooperation is anticipated here.

In working toward their eventual goal of the specification of the LTPL itself, the committee has produced two different types of documents. First, was a set of functional and other requirements which the finished language should satisfy. Second, was a series of definitive discussions of and preliminary specifications for specific aspects or features of the language which the author was assigned to investigate or felt were necessary capabilities for it. Examples of both are included here. Because of space limitations, no attempt has been made to be exhaustive of the work or even of any particular aspect of it. Much greater details will be found in the Minutes of the Workshop or, even better, in the original committee documents themselves.

SECTION I

LONG TERM PROCEDURAL LANGUAGES COMMITTEE WORKING PAPERS ON REQUIREMENTS FOR THE LONG TERM PROCEDURAL LANGUAGE

The following documents on overall desired language features and language requirements are derived from the several *Minutes of the Purdue Workshop on Standardization of Industrial Computer Languages* as follows:

1. "Objectives and Environmental Definition", *Second Minutes*, Chapter VI, pp. 64-73.
2. "Supplemental Report on Requirements", *Second Minutes*, Appendix III, pp. 91-109.
3. "General Specifications of the Language", *Third Minutes*, Chapter VI, pp. 91-110.
4. "Preliminary Description of Chosen Functions-", *Fourth Minutes*, Chapter VI, pp. 103-139.
5. "Summary of Preferable Features-", *Fifth Minutes*, pp. 207, 218-222, by H. Kuwahara of Hitachi, Ltd.

PRECEDING PAGE BLANK-NOT FILMED

OBJECTIVES AND ENVIRONMENT DEFINITION
LONG TERM PROCEDURAL LANGUAGE

OBJECTIVES

The Committee on Long Term Procedural Languages has defined as its objectives:

1. To provide the User Community with a general purpose procedural programming language for industrial computers.
2. To identify means of implementing this language and to promote its implementation.

In other words, this Committee is to identify or design a Procedural Programming Language suited for use with industrial computers. Our work will be successful only if the language's use results in reduced cost to the user and reduced programming time. A prerequisite, moreover, is that the language which it selects be implemented and used. For that reason, the Committee recognizes its responsibility to promote implementation and acceptance of the language as a standard. It also recognizes and states explicitly that it cannot be an implementing group itself.

In doing its work, the Committee has accepted as a point of departure for the report of the Procedural Language Committee of the first Workshop, and the proposals and comments received by the Workshop from attending companies.

PRECEDING PAGE BLANK-NOT FILMED

Development Phases

The Committee's work plan has been broken down into four phases:

1. Definition of the environment in which the language will be used.
2. Functional description of the language.
3. Definition of the language.
4. Promoting implementation and acceptance of the language as a standard.

The general schedule and work outline for each phase are as follows:

1. Environment Definition

   The definition of the language's environment is to yield information that will lead to a functional description of the language's features. This environment definition must include such considerations as:

   a) the nature of the general problem to which the language is addressed
   b) the types of users of the language
   c) the relationship of this language to languages resulting from other Workshop activities
   d) the requirements for compiling the language.

   Attached to this report is the proposed definition of the environment to which our language specification will be addressed.

2. Functional Description of the Language

This effort will concern itself with the general language features which must be made available to the user for programming his problems. Some examples of what should be defined in this effort are:

a) the general type of data communication needed

b) various data types required

c) requirements for supporting run-time software packages.

Also to be considered are such topics as requirements for interfaces with other languages and operating systems. The functional description of the language should be complete by the end of the first quarter of 1970.

3. Language Definition

With a functional description complete, the definition of the language itself can then proceed. An existing language may be taken as a departure point for the language definition, but the Committee will not feel constrained to look only at existing languages and their extensions. The Committee does, however, recognize the merit, from an implementation standpoint, of developing a dialect of an existing, accepted language. The definitions of the language and compiler features are to be completed by the end of 1970.

4. Promoting Implementation and Acceptance of the Language as a Standard

Promotion of the language implementation and acceptance as a standard will be begun as the language definition

effort enters its final phases. This promotion will consist of such activities as:

a) disseminating, as widely as possible, information about the proposed language and its merits

b) encouraging implementing groups

c) generally publicizing and promoting the work, both inside and outside the Committee members' parent organizations.

d) promotion of its acceptance by a relevant standards organization.

ENVIRONMENT DEFINITION

Nature of the Problem

The language is to be designed specifically for real-time applications of industrial computers. This implies

a) the basic ability to express the real-time structure of programs and

b) achievement of the correct balance between flexibility, run-time efficiency and program integrity which is unique to the requirements of the real-time application of industrial computers.

Figure 1 illustrates the objective in terms of the components of software which are required for a typical application.

The two inner circles represent the computer hardware augmented by a minimal kernel of software needed to support programs generated through the language compiler; this is special to the machine.

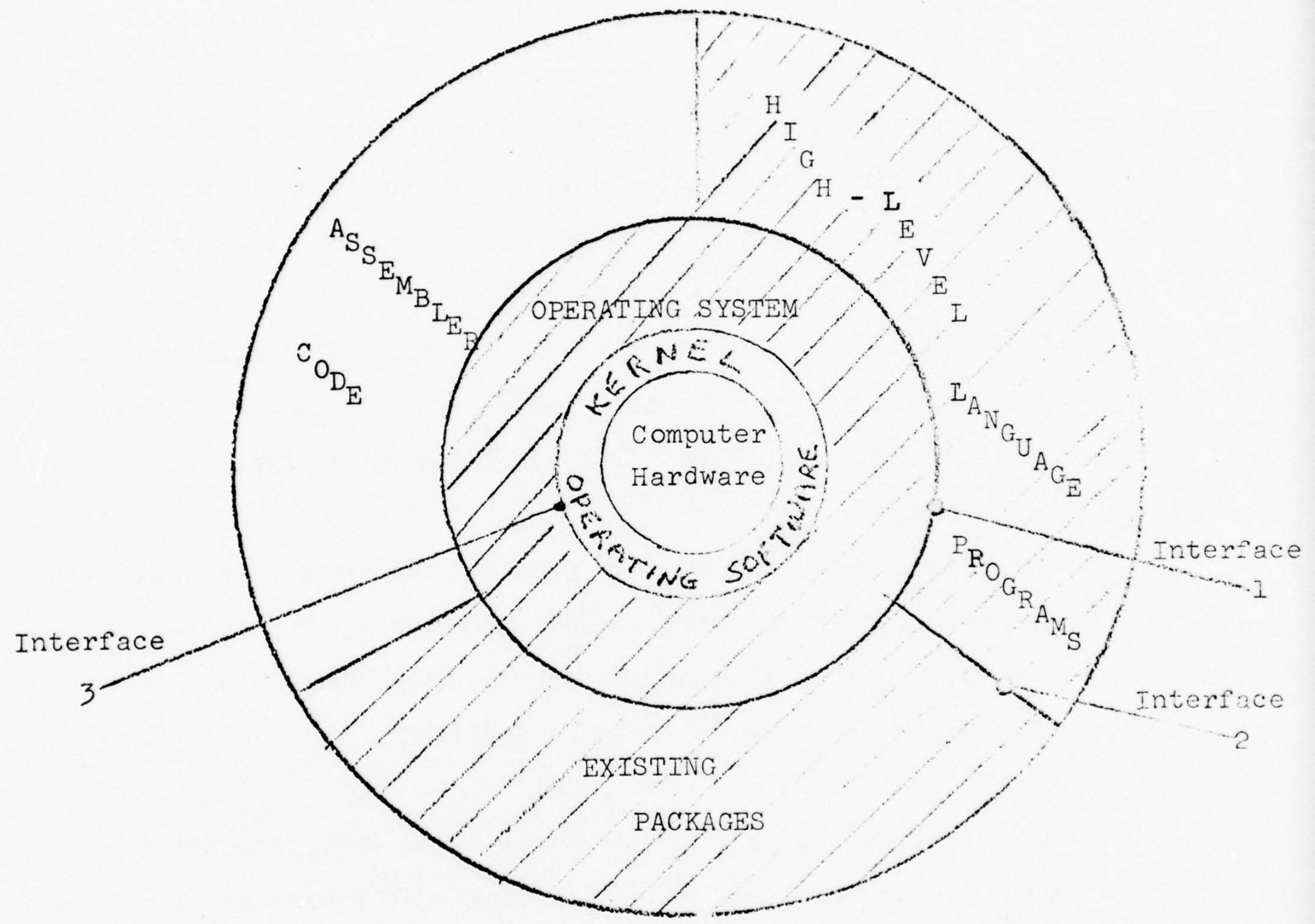


FIGURE 1

Moving outwards from the centre, the next area represents the applications support software generally written by systems programmers and is comparable with what is presently referred to as the operating system. It is primarily concerned with organizing the internal resources of the computer. The outer boundary of this area represents a software interface which provides the support frame-work required by applications programmers.

The outer ring contains the software needed for the particular application, which in general will consist of three components;

a) specially written assembly code programs
b) specially written programs in a high level language; and
c) existing packages, some of which may be standard (including those which implement "fill-in-the-blanks" types of language).

The prime requirement in this area is to provide the applications programmer with security against endangering the system as a whole through programming errors;the language must have a structure which allows this security to be achieved.

Types of Users

There will be two general users for which this language is intended: Systems Programmers and Applications Programmers. These two groups require some differing features in the language.

For Systems Programmers:

a) No operating system can be assumed to exist.
b) The language must be suited to the implementation of such software; namely, it must be a language in which one could write an operating system, a file system, or other "system software".

c) To this end it must make the full hardware capabilities of the computer available to the programmer.

For the Applications Programmer:

a) The typical user will wish a language which has as natural a form as possible.

b) Such systems software features as are required in a general application programming job must be defined within the language. For example, those operating system facilities required for the application software in a multiprogramming environment must be defined explicitly as part of the language.

c) Full knowledge of the target machine need not be necessary to the applications programmer; in fact, the language must reduce to a minimum the need for specific knowledge of the computer hardware.

Relationships to Other Languages

Figure 1 also illustrates three major software interfaces:

Interface 1 - The interface between the high level language programs and the operating system routines which are necessary to support these programs.

Interface 2 - The interface between the pre-programmed applications packages and high-level language programs.

Interface 3 - The interface between the kernel and the operating system.

In the first case, those operating system functions required to support the applications set of this language must be defined, and this constitutes the definition of that interface (but not the method of implementing those functions).

In the second case, the ability must be present to create the interface by use of the systems-programming set of the language.

In the third case, the nature of the kernel and the functions which exist in it will define this interface and therefore this kernel must be defined.

Compiler Requirements

Three criteria which measure the usefulness of the language and its implementation are:

power of expression (or flexibility)

run-time integrity

run-time efficiency.

These factors will often conflict and compromise will be necessary. Considering Figure 2, all language features must be evaluated with regard to their location in the triangle. It is evident that the nature of real-time computing requires that greater weight be placed on the aspect of run-time integrity than in previous languages.

A compiler for the language need not necessarily run in the target machine. However, the full set of the language should be compilable in what is considered a medium-scale industrial computer with some bulk memory. (Speed of compilation will be a secondary consideration in evaluation of the compiler.)

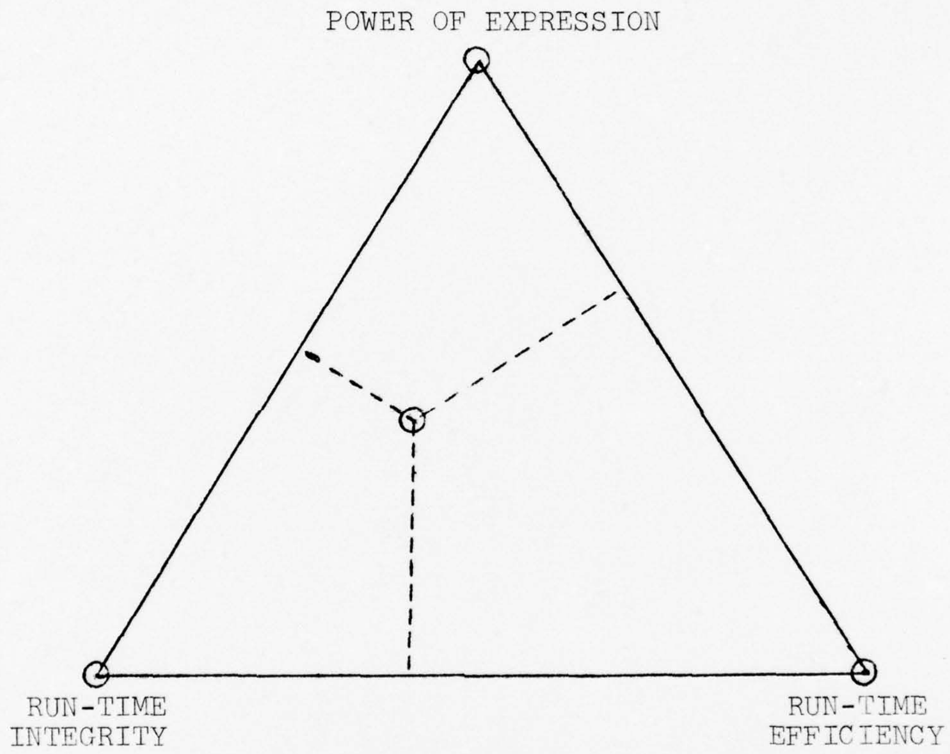


FIGURE 2

Committee notes that discussion of this document before the Workshop indicates that further clarification of the distinction between high-level language application programs and operating systems is necessary.

SUPPLEMENTAL REPORT ON REQUIREMENTS
LONG TERM PROCEDURAL LANGUAGE

INTRODUCTION

This appendix to the Workshop minutes is provided because of the general interest evoked when the Long-Term Procedural Language Committee discussed some of its preliminary thinking in the areas of:

I Data

II Algorithmic Capabilities

III Job Management

IV Compiler Features

Please note that these are preliminary ideas and are, therefore, neither complete nor necessarily compatible. They are instead meant as points for discussion. (Note that other publication of this information is contrary to Workshop rules, since it was not voted on.)

I. DATA CHARACTERISTICS

A variety of characteristics of data, based on the recommendations of the previous Workshop, and study of the language requirements, have been discussed. This note represents the conclusions of the subcommittee to date. It is possible that other data types will be added as a result of further study. The discussion is divided into three main areas:

1. Elementary Classes (floating point, integer, etc.)
2. Data Structures (such as arrays)
3. Inter-program Data Communication (e.g. Fortran Common)

Elementary Classes

For purposes of discussion we have found it useful to consider four classes of elementary data; Numerical, Sets, Program Control, and Peripheral Variables.

Numerical Data

From a language viewpoint, there are two basic types of numbers: whole numbers (integers) and approximations to real numbers (floating point and fixed point). Treating integers as special cases of fixed point data was earlier considered, but rejected because it was inconsistent with the recommended implementation for fixed point. It is recommended that fixed point data be included, provided that a general notation for approximations to real numbers is:

$$m.b^c$$

where m: is the mantissa $0 \leq |m| \leq 1$

b: base (usually 2 or 10)

c: characteristic (an integer)

Floating point machine representations of real numbers consists of a two-part data, one part representing m and the other part representing c, with the value of b usually assumed to be 2, although 10 is assumed on some computers. In the case of fixed point, the value of c is a fixed value, but c varies in floating point numbers.

Most computers in the range of interest do not include floating point hardware, or only as an option. It is expected that this will be an important consideration for several years hence; even

though the availability of floating point hardware will increase. In any case, fixed point arithmetic usually has a significant speed-of-execution advantage over floating point, at the assembly language level, providing the range of values of variables is bounded and does not vary by more than two or three powers of ten.

The associated algorithmic capabilities are simple enough to permit compilation into simple instruction sequences comparable to those normally used today in assembly language. Preliminary investigation shows this is feasible, using the representation described above. In that representation, the value of c is commonly termed the "scale factor".

It is recommended that the proposed language include three numerical data types:

Integer

Floating

Fixed

Complex data has been considered, but there does not appear to be a significant need for this type in industrial computer usage.

Set Data

Datum in this class are values which represent elements of sets. The most familiar examples are logical or boolean data, and character sets. The subcommittee recommends four types of data in this class:

Logical

Character

Status

Bit String

It is worth noting that the main problem with character data is not the character set letter, numbers, etc., but the variety of binary codes used to represent the characters, and the linguistic problems of manipulating the various codes.

Status variables are "borrowed" from the Jovial group of languages, but are novel in the industrial computer area. Their use can best be illustrated by an example:

A variable Alarm Status can be declared to be a status variable, having the states HI, LO, NORMAL. Then in the statements of the language, the following would be typical uses:

a. Alarm Status = LO

b. If Alarm Status = NORMAL then . . . .

Obviously, the same computations can be written with numerical or logical data, but the meaning is much clearer using status variables.

The possibility of treating logical data as a special case of status variables is under consideration.

Bit strings have been proposed, although not completely accepted, because of the tendency to make programs too machine dependent. A bit string is a variable containing a declared number of bits, ordered as a one-dimensional vector. The whole string, substrings of contiguous bits, or single bits may be referenced.

Program Control

Program control data are used to affect the sequence of statement execution. The most common example of such data are statement

labels. The current recommendation is that label variables be provided, as a special case of status variables, i.e., the actual labels which represent the values of the label variable must be explicitly shown in the declaration.

The ability to associate names with interrupts is required, although this fact is recognized in only a few of the examples of industrial control languages. Interrupt Names identify a set of statements which constitute the interrupt response. A class of Interrupt Variables are also required, to permit the definition of more than a single response (but not simultaneously) to a specific interrupt. The requirement for interrupt variables is analogous to the requirement for label variables.

(The topic of a class of address variables, required in the systems subset, has also been discussed. This suggestion was rejected in favor of an operator technique, e.g., if X is a variable, then ADDR (X) produces the address at which X is stored.)

Peripheral Variables

These are a special type of data used in the systems subset of the language. Peripheral variables are names for hardware registers, indicators, and I/O channels, etc., as required to interface between hardware and programs in a higher level language.

Data Structures

As used herein, a data structure is a named set of elementary data. Rectangular arrays are the most familiar examples of data structures. Two kinds are recommended: arrays and structures.

All the elements of an array are of the same elementary data type (homogeneous), and arranged in a rectangular format, in one, two, or three dimensions (subscripts).

A basic structure may contain different types of elementary data, or arrays. For example, a structure with the name DATA1 may contain two integer variables ID and JD, two floating point variables, X and Y, and an array of fixed point variables F (10,2). Structures may contain basic structures. Further investigation of structures, especially as implemented in PL/1, is continuing.

PROGRAM TO PROGRAM COMMUNICATION

The purpose of program to program communication is to transmit data such as set points, limit and process values between programs and to relay logical information such as "set up sequence A, B, C....." or activate devices "X, Y, Z......".

There are basically three ways and two media available for accomplishing this transmission in a single processor system. The media are, obviously, core and disc memory, since, at present, there are no other media available with low enough access time to satisfy system requirements.

Transfers of data within core can take place through local parameter strings or through global common or compool areas available to all programs.

Passing data through local parameter strings has the disadvantage of holding the program containing the string in core until the requesting program has finished processing the data. One way to avoid this problem is to place the parameter strings in a common

core area; but this is effectively the same as global common. Another approach would be to have the program which uses the data transfer the parameter string immediately to its own local buffer area, thus freeing the transmitting program from core. The advantage of passing data in parameter strings is that it allows program initialization requests to closely resemble FORTRAN CALL statements. This is a dubious advantage.

A global common core area is both necessary and desirable. Presumably, variables would be placed in this area as the result of being named in a GLOBAL COMMON declaration. There are two major protection schemes around such an unprotected area, including the proper sequencing of the accesses. For instance, if program A deposits Value X in the common pool and program B fetches it, what is to prevent either program B from fetching it before it is stored, or program C from changing it in the meanwhile. This problem is usually left as an exercise for the ingenious programmer. A well-though-out arrangement of status flags and checks within the operating system can solve this problem. Another thing to consider is partioning of Global Common into subsets, each of which is available to a specific set of programs and to no other. I'm not sure this isn't more trouble than it's worth.

Disc or drum storage provides a means for relieving the core-loading problem at the expense of speed. This is no problem in the transmission of data, say, to a typewriter, but it would not be satisfactory for a high speed data link. Still, drum or disc provides the most common medium for program to program communication.

Generally, the data stored on the disc is in the form of structured files, however, it is conceivable that an unstructured data area such as the Global Common table could spill over onto the discs. Accessing of cammon data on disc has the same sequencing problems that we found in Global Common. This has been solved to some extent by the addition of "private" and "public" attributes to each file, insuring, at least in some cases, that the number of programs with access to a given file is limited. Certainly a system of flags and checks ("busy", "not busy", "locked", "closed", etc.) would be appropriate here as in the core common area. This, again, is a function of either the clever programmer or the operating system.

Because of the intimacy of programs in a process control system and the frequency of communications between them, I do not think it is necessary or feasible to establish elaborate handshaking rituals. In most cases, simple tests should be made available for determining the status and availability of common data.

Communication could be performed through READ/WRITE or INPUT/OUTPUT statements. The unit identifiers could be used to specify more than one core or disc area or perhaps several discs or auxiliary cores. The data string would be unformatted. There appears to be no advantage in formatting data passed between programs and it has the disadvantage of wasting time.

The use of an active statement such as READ or OUTPUT rather than simply referring to a global variable in a statement will allow control of accessing by the operating system. It also provides an easy method for file protection in case this is desirable.

II. ALGORITHMIC CAPABILITIES

A. General

1. The following arithmetic operations should be implemented in the Procedural Language:

   Addition
   Subtraction
   Multiplication
   Division
   Exponentiation

   Parenthesis could define priority of operation as in FORTRAN.

2. The following types of arithmetic operations should be supported:

   Fixed Point
   Floating Point
   Integer
   Boolean
   Mixed Code

   Fixed point arithmetic could be implemented by defining the binary point of each variable and also of every sub-result, i.e.

   A@6 = B@1 + (C@3/D@5)@4
   or A = (B@1 + (C@3/D@5)@4)@6

   This should produce a result in A at a binary point of 6. Mixed Mode arithmetic must be explicit, i.e.

   K = FLOAT ((FIX (X + IDX))/J)

   Where X is assumed to be a floating point number and IDX and J integers. X + IDX would be first calculated and converted to a fixed point number divided by J and the result would be converted to a floating point number to be stored in K.

3. The following functions should be a part of the Procedural Language:

   Square Root
   Sine
   Cosine
   Maximum/Minimum Determination
   Absolute Value
   Arctan
   Truncation Control
   Address Function
   Odd and Even Test
   Remainder from division

A = B = 2X + D

should be implemented. The operation 2X + D would be stored in both A and B.

The following should all be a standard part of the Procedural Language:

4. Relational Operators

   Less than
   Less Than or Equal To
   Equal To
   Not Equal To
   Greater Than
   Greater Than or Equal To

5. Logical Constants

   True
   False

6. Logical Operators

   Shift
   Mask
   Or
   And
   Not
   Exclusive Or

B. Logging and Reporting

For efficient logging and reporting the following features should be included:

1. The ability to send formatted output to any device or read formatted input from any device.
2. The capability of writing "END OF FILE" on any files on any device.
3. Error return to calling program in case a device fails, i.e. WRITE (5, 200, ER1) where:

   5 is the device number,
   200 a Format Number and
   ER1 an error return if device 5 fails.

C. Control Statements

The following control statements or their equivalents should be part of the Procedural Language:

1. GO TO 1 (transfer)
2. GO TO (1, 2, 3, 4 ......), I (computed transfer)
3. DO for loop control
4. IF (expression) 1, 2, 3
5. CALL and RETURN for subroutines
6. CALL MACRO
7. Combinations of the above.

III. JOB MANAGEMENT

A. INTERRUPT RESPONSE PROCEDURES

1. CONNECT (P1, P2)

   P1 - interrupt name

   P2 - routine name

   DISCONNECT (P1)

   P1 = interrupt name

   Allows the programmer to establish a linkage between the interrupt (physical) and his program.

   Note: Amount of pre-processing done by OS must be clearly spelled out.

   Disconnect allows a process but ignores conditions.

2. ARM (P1)

   Allows an interrupt (hardware) to be established.

   DISARM (P1)

   An interrupt (hardware) cannot be established.

   Note: Once an interrupt is established, it remains established until processed.

3. ENABLE (P1)

   Allows the processing of an established interrupt.

   DISABLE (P1)

   Disallows the processing of an established interrupt.

4. Statement to the compiler to specify that the following program segment is an interrupt response routine and to impl certain restrictions upon the program (restricts the segment to a subset).

The labels to be connected to the interrupts may be specified as parameters to the statement signifying an interrupt response routine (this allows compiler to generate special code for interrupt handlers at this level).

Example: Interrupt Block (P1, Pa . . . . . . PN)
P1 .... PN = labels

B. ABILITY TO CALL SUBPROGRAMS IN OTHER LANGUAGES

A statement at the beginning of the program to specify to the compiler what subprograms in this program are referenced by other languages and what these languages are. The other languages to be provided with linkage are at the discretion of the vendor.

Example:		
	Fortran	(sqrt, Sin)
	Algol	(prog 1, prog 2)
	Cobol	(edit, list)
	Assembler	(prog 1, prog 2)
	PL/1	(prog 1, prog 2)
	RTL	(prog 1, prog 2)

C. ERROR DETECTION

ERROR TYPES

Hardware	Software
CPU . . . parity, power failure, protection violation	Program protect
Process Peripherals . . . overloads, function failures, buffer failure	Parameter errors, software drivers busy ex: queue full
DP Peripherals . . function failures, buffer failure	Format error, illegal device no., illegal operation, retry counts

D. REAL-TIME FACILITIES AND PROCESS TASK SUPERVISION

1. schedule - initiation of non-active task from beginning
2. delay - inhibit execution of active task by time segment
3. time schedule - schedule task by time of day
4. suspend - temporarily inhibits execution of task
5. unsuspend - reactivates suspended task
6. non-schedulable - permanent inhibiting of task scheduling (operator intervention required)
7. turn off - sets task to non-active (exit)
8. check status - snapshot of other task state
9. priority - establish a priority level of execution of a task
10. check point - leave task to enter scheduler (task still active)

REAL-TIME FACILITY INTER-ACTIONS OF THE ABOVE

Interrupt Level (call from)	Executing Task (call from)	Other Tasks (effect)	Executing Task (effect)
1. yes	yes	yes	no
2. no	yes	no	yes
3. yes	yes	yes	yes
4. no	yes	no	yes
5. yes	yes	yes	no
6. yes	yes	yes	yes
7. no	yes	no	yes
8. yes	yes	no	no
9. yes	yes	yes	yes
10. no	yes	yes	yes

E. MEMORY ALLOCATION

The ability to reserve and release computer storage at execution time as well as compile time for both program and data should be provided. This implies that statements are needed in the language to not only specify storage at compile time but statements that are executable at run time with variable allocation abilities.

Examples: PL/I statements

Static storage is assigned when a task is loaded, and remains assigned until end of a task.

Automatic storage is assigned on entry to the external subprogram in which it is declared, and is released on exit from that subprogram.

Controlled and Based storage are assigned when during execution of a task it requests the allocation of storage. It is desirable to use these statements across task levels.

Reason:

Need for additional storage only apparent when input data has been supplied.

F. PROGRAM STRUCTURES

Subroutines
Functions - subroutine should be able to know number of parameters that are passed in call

Subprogram functions - order independent parameter lists

Overlays

Interprogram Communication - look at jovial common - named

Common Parameters

V. COMPILER FEATURES

The three basic criteria which the design and implementation of this compiler should attempt to maximize are:

1. Convenience and utility for the programmer.
2. Error detection and reporting.
3. Self documenting aspects.

A. Basic Assumptions and Features

Perhaps the most basic assumption in the language design and compiler implementation should be the explicit declaration and manipulation of all variables and identifiers. This means that all variables, control labels, parameter types, subroutines data types and files, etc., must be explicitly declared before use, and the language will allow no implied types or declaration by use. This feature would allow the compiler maximum ability to detect and report programmer errors.

The compiler implementation must also include a complete set of format or form control options, initiated by reserved control statements and/or control characters. This set would include such commands as * NEW PAGE, * SPACE and, * PAGE HEADER In the same vein, the language should be entirely free format without any fixed column spacing necessary for field registration. Similarly, comments must also be entirely free format (e.g., perhaps enclosed and delineated by $'s).

B. Format Features

The listing generated by the compiler must have compiler generated sequence numbers attached to each line of source code in order to assist in presenting compilation information. Optionally, by control statement, the generated machine code should be available as interspersed printout between the lines of source card images.

Error messages must occur in the printout at the point in which the error was detected and should be of English word type (i.e., no numeric error codes). The detection of an error in a source card should trigger on the printout, the error message along with one or more asterisks over the card column(s) in which the compiler detected the syntactical error(s). This feature would minimize the programmer's time in deducing the nature of a given programming error, particularly in the early stages of programming in the language.

A complete symbol table of all identifiers used should be printed out at the end of the listing, with all names identified as to type, usage and originating statement number. Additionally, an exhaustive cross reference table should be available as an optional printout. In addition to the previous information, this cross reference table would include the sequence numbers of all statements where each particular variable or identifier is mentioned, along with the initial value (if any)

of this variable or identifier. In this symbol table, all identifiers should be grouped according to source usage (e.g., by subroutine and subprogram block, by data file etc.).

C. Language Related Features

Another major language feature which has great implications for the compiler implementation is that of compile time statements. This feature would allow the use of some subset of the complete language as compile time commands by enclosing the desired data and/or operations in square brackets (e.g., [...]). This feature would allow such usage as conditional compiling and the use of compile time variables as parameters or (equivalently) run time constants. Thus, a number of data areas and/or arrays could be compiled with different sizes by changing one compile time variable (which is contained in each of the declaration statements).

The language and the compiler should allow system declarations (i.e., have an INCLUDE statement). This feature would support such usage as defining a set of system symbols for the basic or skeleton executive calls, and including these in all subsequent programs which reference the executive by the use of the INCLUDE statement. In addition to such an equivalence table, the statement could also be employed to minimize program preparation in routines referencing data bases, compools, named commons, or equivalent concepts. As an example of the use of this concept, a large number of programs could reference

and manipulate the records constituting a major DDC data base, with the necessity of having to define and keypunch these declaration cards in one program only.

D. Implementation

A compiler should be implemented in some easily transportable form, so that a wide range of users will have access to the compiler on whatever machines are available to them. This is not to say that the compiler must run on a 4K core memory mini-computer, but rather on a wide range of larger, widely available machines such as the typically ubiquitous DP machines. This implementation may possibly take the form of a compiler-compiler generator or implementation in a meta language.

Two major questions arise in any discussion of implementation, which must be definitively answered before any plans are made. The first of these asks the question whether it is desired to have a unique compiler for each desired target machine, or should one compiler be capable of compiling source programs for a number of target machines.

The second question concerns whether or not it is desired to have definite subsets of the language available in one implementation. For example, will there be a definite systems programmer subset and user application subset available under one compiler. In summary are we talking about one compiler, two compilers, or many compilers, and of what capabilities?

GENERAL SPECIFICATION OF THE LANGUAGE

(Working Paper - Not approved as yet by the Workshop)

A. Basic Design Assumptions

This document specifies the requirements for a language for the programming of industrial control computers. This language is intended for both the applications and system's programmers usage, with the primary emphasis being given to the former. Application programs are those which apply to specific applications and are somewhat machine independent. Systems programs have to do with the operation of the computer independent of the specific application and are largely machine-dependent.

These dual objectives are to be achieved by defining two versions of the language (which may in practice be one language with two subsets if the intersection of the two is sufficiently large) with commonality as a secondary objective.

The implementation of a compiler for the language need not necessarily be possible on a given configuration of the target machine. However, the compiler must be able to produce programs which will execute in any viable configuration of the target machine. The implementation of a compiler for the full language set should be possible on what is currently considered to be a medium scale configuration of an industrial computer. Typically this would include 16k of core memory, disc or drum bulk memory and some form of high speed data processing input/output.

PRECEDING PAGE BLANK-NOT FILMED

B. Form of the Language

1. Character set

The character set should be simple enough to be common to most data processing devices and familiar to most programmers. The recommended set should be expressible in a 6 bit code.

2. Identifier definition

Identifiers are names of programs and entities within programs. These identifiers may be manipulated at run-time or compile-time, as appropriate.

a. Formation Rules

The formation rules should be as minimally constraining as possible. Allowing an indefinite number of characters in an identifier is in the direction of greater flexibility, although a fixed number might be acceptable.

b. Use of Reserved Words

Certain words will be defined as reserved words. A reserved word may not be used as an identifier.

c. Data Names for Aggregates

Subscripting will be used for such data entities as required (strings, arrays). The method of qualified names will be employed for data structures. For example, if T101 is the name of a set or collection of items and SETPOINT is the name of a particular data item in that set, then it can be referenced as T101 SETPOINT.

3. Definition and Usage of Basic Elements

a. Operators

At least the following classes of operators are included: arithmetic, relational, logical (Boolean); and string operators. There will also be some ability to operate on address variables.

b. Delimiters

The delimiters actually employed to separate and terminate syntactical entities are details of the actual language design. However, "card column" placement will not be used as a delimiter.

c. Significance of Blanks

Blanks are significant delimiters, but a sequence of blanks has the same syntactical significance as a single blank.

d. Comments

The ability to insert comment text anywhere is extremely useful for documentation. Comment text will be preceded and followed by a specified special character, or characters, and the entire sequence will have the same significance as a blank.

e. Literals

Literals will be provided as appropriate to data types and provision of decimal, octal, hexadecimal and binary constants will be included.

f. Type of Input Form

The conceptual input form will be logical input records (i.e. statements) and these logical records shall be independent of the physical input form of media.

C. Structure of Program

1. Types of Program Structure

We distinguish two types of program structure, structure which is related to the compile time organization of code and structure which is part of the run-time organization of code.

2. Compile-Time Structure

a. Types of Units

The smallest executable unit in the language is the simple statement.

A block is a grouping of statements with some declarations.

A procedure is a named block. A program module is a collection of statements which can be compiled as unit.

b. Scope of names

(1) Identifiers.

An identifier has the scope of the block in which it is declared unless it is declared global in which case its scope is the entire system.

(2) Procedures

The scope of procedure names is like that of identifiers except that procedures declared in the outermost block of a program module are assumed to be external and have global scope.

(3) Scope of Labels

To maintain program integrity, transfers should be permitted only to explicitly labelled points within the procedure which contains the transfer. This forbids transfers out of nested procedures and transfers to labels within parameter lists. The same effect can be achieved by alternate procedure exits.

3. Run-Time Structure

a. Types of units

A major program is a body of coded instructions activated by or through the environment. The environment is defined as the process, computer, and the standard operating system. A subprogram is a body of code entered from a major program or other subprogram which returns control through the invoking major program or subprogram. A task is a specific 'run-time' execution of a major program and its subprograms. A major program or a subprogram may be reentrant; this is not a language requirement, however.

b. Interaction between Major Programs and Subprograms Communication between invoking programs (either major programs or subprograms) and the invoked program (subprogram) can occur in 3 ways. (1) By sharing data areas (only if both components have been compiled together in a single program module), (2) by referencing global data areas, and (3) by explicit passage of parameters from the invoking program to the subprogram. The subprogram may return values to the subprogram. The subprogram may return values to the invoking program upon its completion in the same way.

e. Interactions between tasks and the operating system. Generally speaking, facilities present in the operating system should appear to the task the same as separately compiled subprograms.

d. Interactions between Tasks

(1) Tasks shall have the ability to activate other tasks, to terminate themselves, and to suspend themselves until the occurrence of an event, external to the task.

(2) In scheduling tasks for activation, data appropriate to the activated tasks may be passed from the task performing the activation in a manner similar to the communication between programs and subprograms.

EXAMPLE PROGRAM STRUCTURE

Program Module

```
declaration M

procedure A
  declarations N

end
procedure B
  declarations P

     procedure C
       declarations R

     end

end
```

Declarations M are available in A, B, C.

Declarations N is available in A.

Declarations P is available in B, C.

Declarations R is available in C.

Procedure A, B are global.

Procedure C is available from B.

D. Data Types and Units and Computations with Them

Data types may have the attributes of base, scale, precision where:

Base: decimal, binary, octal, or hexadecimal

Scale: Fixed, float

Precision: the minimum number of decimal digits to be maintained and a scale factor.

Data types are declared either as named variables thru declarations or as data constants via use in context of the program.

In either case the representation must be sufficient to define all the attributes necessary for that data type as described below.

1. Types of Data Variables and Constants

a. Arithmetic data

(1) Fixed point - A fixed point datum is represented by one or more decimal digits with an optional decimal point, followed by the letter S, followed by an optionally signed scale factor, followed by a minimum precision attribute.

If the decimal point, the letter S, and the scale factor are not present then the constant is an integer constant.

Integers are represented with a scale factor of zero

(2) Floating point data - A floating point datum is represented by one or more decimal digits with an optional decimal point or may be represented by one

or more decimal digits with a decimal point; followed by the letter E, followed by an optionally signed exponent followed by a minimum precision attribute. The exponent is one or more decimal digits specifying an integral power of ten.

b. Boolean data

Data possesses only two states; true/false

c. String data

(1) Bit string consists of a string of zero or more bits.

(2) Character string consists of a string of zero or more characters in the data character set.

d. Locator Variable

(1) A declared variable whose value is a pointer to one of the various declared data elements accessible to the procedure.

e. Hierarchical

(1) Arrays - an array is a multi-dimensional, ordered, named collection of elements, all of which have identical data attributes.

(2) Data Structure - a named hierarchical collection of scalars, arrays, and structures. These need not be of the same data type nor have the same attributes.

Example of Named Data Structure

```
1 A,
    2 B,
    2 C,
        3 D(2),
        3 E,
    2 F;
```

This example is taken from PL/I. The data structure named "A" is substructured into three other data structures ("B", "C" and "F"). Substructure "C" is further substructured into "D" (which is an array) and "E".

Reference to "A" accesses the entire structure. Reference to "B" accesses B only, and reference to "C" accesses "D" and "E" as a structure.

f. Status

Set of program declared states. Each state in the set has an associated identifier defined by the programmer. The value of a status variable is one of the states in the set, and this value may be changed or tested in the program.

g. Label

A variable, whose value is one of the labels of statements defined in the procedure.

2. Types of Arithmetic

a. Arithmetic Operator

(1) Addition, subtraction, multiplication, division, exponentiation.

(2) Re-scale fixed point data, i.e., arithmetic shift.

b. Multiple precision

This is achieved thru specification of the minimum precision for the data types specified previously.

e. Boolean operations (on boolean types and bit string types)

(1) Negation, and, or, and exclusive or

d. String operation

(1) Comparing, assembling and shifting

e. Higher level operations such as matrix multiplies, etc., involving arrays and data structures are illegal as implicit operations. Boolean operations on bit strings are permitted.

3. Rules on Creation and Evaluation of Arithmetic Expressions

a. Intermingling of data types is not permitted implicitly.

b. Conversions of data types is accomplished thru explicit operations.

c. Precedence rules will be applied in the evaluation of arithmetic expressions.

E. Executable Statements

1. Assignment Statements

This type of statement has the capability of assigning a value to a data element. The language will have an assignment statement.

2. Alphanumeric Data Handling

These operations involve the handling of alpha-numeric data for use both inside and outside the computer by editing and conversion, and sorting statements. Editing statements as defined here prepare data for use outside the computer.

defined here prepare data for use outside the computer. Conversion statements prepare data for internal use. The language will provide editing and conversion capability. Sorting statements were considered and will not be included in the language.

3 Program Control Statements

a. Control Statements where scope is within a block (GO TO, IF, DO,) will be not unlike the PL/1 definitions.

b. Decision tables have been considered and will not be included in the language.

4 Symbolic Data Handling

This type of facility provides for operations on strings of symbolic characters - operations such as concatenate, append, or other functions. These facilities are typical in languages such as LISP and SNOBOL.

The language being defined will not have facilities of this kind, also see E.2 and C.2.d for related facilities.

5 Interaction with operating system and/or equipment.

Error conditions: facilities are to be provided so that handling of errors can be programmed in the language but the language should provide for defaults if explicit error handling is not included in the program.

a. I/O Statements

The language will support I/O statements appropriate for physical devices and for logical files. Both files and devices may be referred to in a symbolic manner

I/O statements should include statements to attach or detach a physical device to the task and statements to open or close a logical file.

Statements with format conversion as well as format free I/O will be supported.

b. Library Reference Statements

A library of routines will be provided. A standard set will be defined. There must be capability for user additions to the standard set, either through additions to the standard library or through providing a different user library facility.

c. Debugging Statements

See Section 7 - Structure of language and compiler interaction.

d. Data Storage Allocation Statements

These are statements allowing a running program to request allocation and deallocation of core or bulk memory at run time. The language will contain statements of this type. Run-time support is an operating system feature, which may not exist.

e. Operating System and Machine Feature Statements

The following statements assume multi-programming and a multi-level interrupt structure.

(1) Activate - initiation of a non-active major program from the beginning with or without parameter passage.

NOTE: Activation of another major program in a manner like a subroutine, ie. return to the calling program after completion of the called major program, can be effected through coding the activated program to accept a parameter which is the identification of the calling program. It will reactivate the calling program upon completion. We therefore recommend this facility not be provided in the language.

(2) Delay - Inhibit execution of the requesting major program by time interval.

(3) Schedule - schedule another major program to be activated at a time of day or after a specified time increment.

(4) Suspend - Temporarily inhibits execution of a task until an external event. See 3.3.4.1

(5) Resume - Undoes a suspend.

(6) Lock-out - inhibits scheduling a major program (Operator intervention required to undo).

(7) Turn off - set calling task to non - active (normal exit).

(8) Check status - enables a task to interrogate status of another task.

(9) Priority - will allow the task to establish a priority level for its or another task's operation.

f. Interrupt response facilities the same or equivalent to those following shall be provided.

(1) CONNECT (P1, P2)

P1 = interrupt name (logical)

P2 = interrupt processing routine name

Allows the programmer to establish a linkage between the interrupt and a designated interrupt servicing program.

DISCONNECT (P1)

P1 = interrupt name (logical)

Allows a process interrupt to occur (depending on arm/enable status) but the calling routine will no longer service the interrupt. The interrupt will not be serviced until a "connect" is made for that interrupt.

(2) ARM (P2) allows a hardware interrupt to be recognized and remembered.

DISARM (P1) causes an interrupt signal to be completely ignored.

NOTE: Once an interrupt is armed it remains armed until disarmed.

(3) ENABLE (P1) - allows the processing of an established interrupt. Disable (P1) - prevents the processing of an "interrupt". This action is negated by an ENABLE and processing of any waiting interrupts will occur.

General notes for E.

All reserved words will be forbidden for use by the programmer in any other meaning or context.

All operations will be checked by the operating system for validity.

F. Declarations and Non-Executable Statements

1. A statement (not unlike DECLARE in PL/1) should be available for defining the characteristics, attributes or value of variables. All variables must be explicitly declared before use, and the language should not allow implied types or declaration by use. This feature would allow the compiler the maximum ability to diagnose and report programming errors.

   Attributes that can be assigned to a variable will depend upon data types the language will have.

2. File Description

   A file is an organized collection of information. The language will provide the ability to define file attributes for user processing requirements. Some of these attributes are:

   a. Device allocation
      (1) Device type (disk, tape, card reader, etc.)
      (2) File size (number of records, tracks, cylinders, etc.)
   b. File type (sequential, random, etc.)
   c. Record Format (fixed, variable)
   d. Record size
   e. Blocking factor
   f. Buffer requirements
   g. File Protection

3. Format Description

A format statement will be provided to convert internal variables into character strings for output, or converting input character strings into internal variable format.
The statement will also control the precision of the conversion as well as the type.
A feature to read in format statements at run-time should not be part of the language, to allow for efficient interpretation of format operations.

4. Storage Allocation

The ability to reserve and release computer storage at execution time as well as compile time for both program and data should be provided. This implies that statements are needed in the language to not only specify storage at compile time but statements that are executable at run time with variable allocation abilities. Declarations for specification of overlay structures will be provided.

Example: Static storage is assigned when a task is loaded, and remains assigned until end of task.

Automatic storage is assigned on entry to the external subprogram in which it is declared, and is released on exit from the subprogram.

Controlled and Based storage are assigned when during execution of a task it requests the allocation of storage. It is desirable to use these statements across task levels.

G. Language and Compiler Interaction

While it is true that the bulk of the details concerning the compiler design and implementation are not the concern of a language specification, there are a number of compiler features which are either implied by the language or whose inclusion makes the use of the language much easier.

1. Self Modification of Programs

A language feature which has great implications for the compiler design is that of compile time statements. This feature would allow the use of some subset of the complete language as compile time commands by enclosing the desired data and/or operations in square brackets (for example). This feature would allow such usage as conditional compilation and the use of compile time variables as run time constants.

2. Self-Extension of the Language

Another concept which has been discussed, but is not included in the language specification, is that of language extendability. This amounts to having the ability in a user program to define and use macros for the generation of a repeatedly used code sequence in line.

3. Ability to Write the Compiler in the Language

This feature is not considered to be a desired design specification.

4. Language Design and Implementation Efficiency

a. Compile time versus run-time

It is assumed that run-time efficiency is more

important than compile time by a couple orders of magnitude.

b. Generality versus restriction

Perhaps the most basic assumption in the language design and compiler implementation should be the explicit declaration and manipulation of all variables and identifiers. This means that all variables, control labels, parameter types, subroutines, data types and files etc., must be explicitly declared before use, and the language will allow no implied types or declaration by use. This feature would allow the compiler maximum ability to detect and report programmer errors.

5. Debugging Aids and Error Checking

The listing generated by the compiler must have compiler generated sequence numbers attached to each line of source code in order to assist in presenting compilation information. Optionally by control statement, the generated machine code should be available as printout with the lines of source statements.

Error messages must occur in the printout at the point in which the error was detected and should be of English word type (i.e. no numeric error codes). The detection of an error in a source card should trigger on the printout, the error message along with one or more asterisks over the card column (s). This feature would minimize the programmer's time in deducing the nature of a given programming

error, particularly in the early stages of programming in the language.

As far as is economically reasonable and possible, errors should not be fatal (i.e., compilations should continue and as much source code generated as possible with words of a pre-defined code generated for bad core locations).

A complete symbol table of all identifiers used should be printed out at the end of the listing, with all names identified as to type, usage and originating statement number. Additionally, an exhaustive cross reference table should be available as an optional printout. In addition to the previous information, this cross reference table would include the sequence numbers of all statements where each particular variable or identifier is mentioned, along with the initial value (if any) of this variable or identifier. In this symbol table, all identifiers should be grouped according to source usage (e.g., by subroutine and sub-program block, by data file, etc.)

Included in the language and its implementation should be available some type of at least minimal data and/or control trace, with ability to initiate and discontinue either type of tracing by in line commands.

PRELIMINARY DESCRIPTION OF CHOSEN FUNCTIONS FOR INCORPORATION INTO PROPOSED SPECIFICATION OF LONG TERM PROCEDURAL LANGUAGE

INTRODUCTION

The task of the Long Term Procedural Language committee was to create a new procedural language, identifying the specific requirements of process control, and excising those features of procedural languages which are superfluous to the process control task. As the committee approached the task of specifying a definite language, it had several courses open to it. It could create an entirely new language or it could choose to alter an existing language to its needs. Because of the sheer amount of work involved in the creation of a new language, and of its poor changes of acceptance, the committee chose the latter course. Of the existing languages available for modification, the committee choose PL/I. FORTRAN would be a likely candidate except that its syntax is too rigid, it does not have sufficient modularity, and it has already become the subject of standardization. ALGOL would be a likely candidate except that it is not widely used in this country, and acceptance of a process control ALGOL might be difficult. JOVIAL also is a candidate, except that it is not widely used outside of the military, and its syntax is somewhat archaic. It seemed to the committee that PL/I was in the right state of development for our purposes. On the one hand, the language is rich enough so that a subset might be expected to serve the needs of process control adequately.

The language is sufficiently stable that good documentation is available, and compilers are starting to exist for machines other than the IBM 360. On the other hand, the language is still new enough so that modifications in PL/I itself are possible. Discussions of standardization are only now beginning, and if process control requirements dictate that some feature of PL/I be changed so as to accommodate process control as a subset, this is still within the realm of possibility.

The largest difference between process control situations and that of FORTRAN or PL/I is not in any given statement or program structure, but in the interpretation of the language with respect to the computer on which it runs. FORTRAN and PL/I both envisage a computer which is totally dedicated to the solution of some particular task described by the programmer, a task which has a well defined beginning and conclusion. This is true even in time-sharing environments, because the aim of time-sharing is to make it appear to each programmer that he has the entire computer at his disposal. In general, the relationship between the time the computer takes to execute a given task, and the nature of the task is irrelevant.

The situation in process control is somewhat different. In general, there is no clear cut termination of a computer run; the computer continues its operations as long as the process it is controlling continues. The relationship between

the times of computer operations and the times involved in the controlled process is significant and perhaps even crucial. In general, a given programmer does not have the entire computer at his disposal; he is sharing it with others whose tasks also have certain response requirements. In short, the process control operation is a real-time operation, and this imposes certain requirements on the language which expresses the operation.

The remainder of this document describes the language the committee has proposed. Generally speaking, we have specified a subset of PL/I, including only those features we considered useful for the process control task. We aimed to create a language somewhat smaller than PL/I so that it would be easier to learn, but more important so that reasonable implementations would be possible on medium scale computers. Many of the automatic features of PL/I have been excised; it was felt that the overhead of implementation of these features could not be tolerated on process control computers. These features include things like automatic storage allocation and general mixed mode conversion. Many features of PL/I are directed to business data processing; these have been eliminated as being irrelevant to the process control task. Finally, in the area of subordinate tasking, several statements have been re-interpreted so as to allow the language to express the real-time problems of process control.

SECTION 1: GENERAL LANGUAGE FEATURES

GENERAL FORM

The general form of the language will be independent of its physical input medium. A program statement will be a string of characters terminated with a special character, the semicolon. The PL/1 character sets will be adopted for the language. In general the maximum length of identifiers will be implementation defined, but must be at least 4.

BASIC PROGRAM STRUCTURE

A program is constructed from basic program elements called statements.

Statements are grouped into larger program-elements, the group and the block. There are two types of statements: simple and compound.

A simple statement is defined as:

[[statement-identifier]

statement-body] ;

The "statement identifier," if it appears, is a keyword characterizing the kind of statement. If it does not appear, and the statement body does appear, then the statement is an assignment statement.

COMPOUND STATEMENTS

A compound statement is a statement that contains other program-elements. There are two types of compound statement:

The IF compound statement

The ON compound statement

Examples:

```
If A-B THEN GO TO S1;  ELSE A+C:

ON OVERFLOW GO TO OVFIX:
```

Each PL/1 statement is described in the alphabetic list of statements in Chapter 8.

LABELS

Statements may be labeled to permit reference to them. A statement label will be an identifier separated from the body of the statement by a colon. A statement may have multiple labels.

EXAMPLE JOE: SAM: A-B ;

PRECEDING PAGE BLANK-NOT FILMED

GROUPS

A group is a collection of one or more statements and is used for control purposes.

A group has one of two forms. The first form, called a DO-group, is:

```
[label:] ...    DO-statement
                program-element-1
                program-element-2

                END   [label]  ;
```

The label following END is one of the labels of the DO statement.

The second form of a group is simply a single statement, as follows:

```
[label] ...   statement
```

The "statement" is any statement except DO, END, PROCEDURE, DECLARE, FORMAT, or ENTRY.

Example of the first form:

```
ALPHA:  DO;

 A+B*C;

 IF A   0 THEN DO;
 B=1;
 C=0;
 END;
 ALPHA:
```

In the example above, any of the single statements --- except the DO and END statements -- is an example of the second form of a group.

BLOCKS

A block is a collection of statements that defines the program region -- or scope -- throughout which an identifier is established as a name. It also is used for control purposes.

There is one kind of block, a procedure block.

SECTION 2: DATA ELEMENTS

Information that is operated on in an object program during execution is called data.

DATA ORGANIZATION

Data may be organized as scalar items (i.e., single data items) or aggregates of data items (i.e., arrays and structures).

SCALAR ITEMS

A data item may be either a constant or the value of a scalar variable. Constants and scalar variables are called scalar items. Scalar variables and scalar data items may also be called element variables and element data items respectively.

Constants

A constant is a data item that denotes itself, i. e., its representation is both its name and its value; thus, it cannot change during the execution of a program.

Scalar Variables

A scalar variable, like a constant, denotes a data item. This data item is called the value of the scalar variable. Unlike a constant, however, a variable may take on more than one value during the execution of a program.

Reference is made to a scalar variable by a name, which may be a simple name, a subscripted name, a qualified name, or a subscripted qualified name.

DATA AGGREGATES

All classes of variable data items may be grouped into arrays or structures. Rules for this grouping are given below.

Arrays

An array is an n-dimensional, ordered collection of elements, all of which have identical data declarations.

The number of dimensions of an array, and the upper and lower bounds of each dimension, are specified by the use of the dimension attribute.

Example:

```
DECLARE   A(3,4);
```

This statement defines A as an array with 2 dimensions: 3 rows and 4 columns.

The elements of an array may be structures (see "Arrays of Structures").

Structures

A structure is a hierarchical collection of scalar variables, arrays, and structures. These need not be of the same data type nor have the same attributes.

The outermost structure is a major structure, and all contained structures are minor structures. Scalar variables or arrays forming the elements of major or minor structures are known as base elements.

A structure is specified by declaring the major structure name and following it with the names of all contained minor structures and elements. Each name is preceded by a level number, which is a non-zero decimal integer constant. A major structure is always at level one and all minor structures and elements contained in a structure (at level n) have a level number that is numerically greater than n, but they need not necessarily be a level n+1, nor need they all have the same level number.

Examples:

```
DECLARE 1 ALOGPT,
          2  CVALUE,
          2  LIMITS,
               3  HILIM,
               3  LOLIM,
          2  ALARMNO;
```

In the above example ALOGPT is defined as the major structure containing the scalar variables CVALUE and ALARMNO and the structure LIMITS. The structure LIMITS contains the scalar variables HILIM and LOLIM.

Arrays of Structures

An array of structures is formed by giving the dimension attribute to a structure.

Attributes of Structures

Structures and arrays of structures are not given data attributes. These can be given only to structure base elements.

Major structure names may be declared with the EXTERNAL attribute. Items contained in structures may not be declared with the EXTERNAL attribute, and even if INTERNAL is unspecified, they are assumed to be INTERNAL. When the same major structure

name is declared with the EXTERNAL attribute in more than one block, the attributes of the structure members must be the same in each case, although the names of the structure members need not be the same. A reference to a member in one such block is effectively a reference to that member in all blocks in which the external name is known, regardless of the names of the members.

NAMING

This section describes the rules for referring to a particular data time, groups of items, arrays, and structures. The permitted types of data names are simple, qualified, subscripted, and subscripted qualified.

SIMPLE NAMES

A simple name is an identifier that refers to a scalar, an array, or a structure.

SUBSCRIPTED NAMES

A subscripted name is used to refer to an element of an array. It is a simple name that has been declared to be the name of an array followed by a list of subscripts. The subscripts are separated by commas and are enclosed in parentheses.

Examples:

```
A (3)
FIELD  (B,C)
PRODUCT (SCOPE * UNIT + VALUE, PERIOD)
ALPHA  (1,2,3,4)
```

QUALIFIED NAMES

A simple name usually refers uniquely to scalar variable, an array, or a structure. However, it is possible for a name to refer to more than one variable, array, or structure if the identically named items are themselves parts of different structures. In order to avoid any ambiguity in referring to these similarly names items, it is necessary to create a unique name; this is done by forming a qualified name.

Examples:

A program may contain the structures:

```
DECLARE 1 ANALOGIN, 2 DESCRIPTION, 2 UNITS, 2 HILIMIT,
2 LOLIMIT;

DECLARE 1 DIGITALIN, 2 DESCRIPTION, 2 UNITS, 2 LIMIT;
```

Elements are then referred to as:

ANALOGIN.UNITS
DIGITALIN.UNITS
ANALOGIN.DESCRIPTION

DATA TYPES

The types of data allowed can be categorized as problem data and program-control data.

PROBLEM DATA

Problem data is any data that can be classified as type arithmetic or type string.

Arithmetic Data

An arithmetic data item is one that has a numeric value with characteristics of base, scale, and precision.

Base (decimal or binary), scale (fixed-point or floating-point), and precision have reference to internal representation of the data described and to the internal arithmetic that is to be used.

Arithmetic Variables

Arithmetic variables are names of arithmetic data items. These names have been given the characteristics (i.e., attributes) of base, scale, and precision.

String Data

String data can be classified as character-string or bit-string. The length of a string data item is equaivalent to the number of characters (for a character-string) or the number of binary digits (for a bit-string) in the item. A string data item of length zero is known as a null string.

Character-String Data

Character-string data consists of a string of zero or more characters in the data character set.

CHARACTER-STRING CONSTANTS: A character-string constant is zero or more characters in the data character set enclosed in quotation marks.

Examples:
'$ 123,45'
'HIGH ALARM'

Bit String Data

Bit-string data consists of a string of zero or more binary digits (0and 1).

BIT-STRING CONSTANTS: A bit-string constant is zero or more binary digits enclosed in quotation marks, followed by the letter B.

Examples:

'0100'B
"101'B

PROGRAM-CONTROL DATA

Program-control data is any data that can be classified as type label, entry, task, event, pointer, offset, or area.

Label Data

Statement-label data is used only in connection with statement labels, and has values which permit references to be made to the statements of a program. Statement label data may be constants or variables.

Statement-Label Constants

A statement-label constant is an identifier that appears in the program as a statement label. It permits references to be made to statements.

Example:

```
ONCEOVER:  IF LENGTH > MINI THEN GO TO BUST;
                 .
                 .
                 .
           GO TO ONCEOVER;
                 .
                 .
                 .
BUST:  RETURN;
```

ONCEOVER and BUST are statement-label constants.

Statement-Label Variables

A statement-label variable is a variable that has as values statement-label constants. These variables may be grouped into arrays, or they may be elements of structures.

Example:

```
                DECLARE X LABEL;
                X = POSROUTINE;
                        .
POSROUTINE:             .
                        .
                X = NEGROUTINE;
                GO TO X;
                        .
NEGROUTINE:             .
                        .
```

The label variable X may have the value of either POSROUTINE or NEGROUTINE, both lables in the procedure. In the above example, GO TO X transfers control to NEGROUTINE.

Entry Data

Entry data is used only in connection with entry points, and has values which permit references to be made to entry points of a program. Entry data may be constants or variables.

Entry Constants

An entry constant is an identifier that appears in the program as an entry name. It permits references to be made to an entry point of a procedure.

Entry Variables

An entry variable is a variable that has as values entry constants. These variables may be grouped in arrays or they may be elements of structures.

Task Data

A task variable is the name of a task. A task variable may be an element of an array or of a structure. The priority associated with a task variable may be assigned in the CALL statement, or in an assignment statement by use of the PRIORITY pseudo-variable.

Event Data

An event variable is the name of an event used in connection with asynchronous processing, in multitasking, or with record-oriented I/O operations. An event variable may be an element of an array or of a structure.

An event variable has associated completion and status values that can be accessed by the COMPLETION and STATUS built-in functions.

Locator Data

Locator data consists of pointer variables and offset variables. A pointer variable has a value that is used to identify the location of a single generation of a variable. An offset variable has a value that is used to identify the location of a based variable relative to the beginning of an area.

Locator Qualification

Locator qualification is used to associate one or more pointer or offset values with a based variable to identify a particular generation of data.

Examples:

```
A = P->B;
A = P->Q->B;
A = ADDR(X)->B;
```

The first example causes assignment to A of the value of B in the generation pointed to by P. The second example specifies that the value of P is to be used to locate the generation of Q which locates the specific generation of B to be assigned to A. In the third example, the generation of B is derived from the location of the variable X.

Area Data

An area variable represents an area of storage in which based variables may be allocated and freed.

SECTION 3: DATA MANIPULATION

EXPRESSIONS

An expression is an algorithm used for computing a value. Syntactically, an expression consists of a constant, a scalar variable, an expression enclosed in parentheses, an expression preceded by a prefix operator, two expressions connected by an infix operator, or a function reference that returns a scalar value. An expression represents a scalar value. Only the operators = and ¬= may appear with label, pointer, and offset data. No operators may appear with entry, area, event, and task data. If A and B are expressions, then the operators + and - used in expressions of the form +A or -A, are called prefix operators. When these operators are used in expressions of the form A+B or A-B they are called infix operators. There are four classes of expressions:

1) Arithmetic
2) comparison
3) bit string and
4) concatenation.

If an operator is present, it determines the class. If there is no operator, the single operand determines the class.

Arithmetic Operations

An arithmetic expression of any complexity is composed of a set of elementary arithmetic operations.
An elementary arithmetic operation has the following general format:

[+ or - operand]

or

operand [+ or - or * or / or **]operand

Only plus and minus may be used as prefix.

The base, scale, mode, and precision of the result depend on the operands and the operator in the following ways:

1. Prefix operations: The prefix operations of plus and minus yield a result having the scale and precision of the operand.

PRECEDING PAGE BLANK-NOT FILMED

2. Floating-point: If the operands of an infix operation are floating-point the result is a floating-point result and is the greater of the precisions of the two operands.

3. Fixed-point: If the operands of an infix operation are fixed the result is fixed. The precision of a fixed-point result depends on the operation and the precisions of the operands.

Bit-String operations

Bit-string operations have the following general forms:

¬ operand
operand & operand
operand | operand

The prefix operation "not" and the infix operations "and" and "or" are specified above. The operands must be bit-string type. The result will be of bit-string type.

Comparison Operations

Comparison operation have the general form-operand

{< or ¬ <or<= or = or ¬= or >= or > or ¬ >} operand

there are three types of comparisons:

1. Arithmetic, which involves signed numeric values in arithmetic form.

2. Character, which involves left-to-right, comparisons of Character strings.

3. Bit, which is left to right comparison of bit strings.

The result is the string '1' B if the relationship is true, '0' B if false.

Concatenation Operations

Concatenation Operation have the following general form:

operand || operand

Both operand must be of the same type. The length is the sum of the two lengths.

PRIORITY OF OPERANDS

Highest: ¬, ** , prefix + , prefix -

*, /

infix +, infix -

||

>= , >, ¬>, ¬= , <, ¬<, <=, =

&

Lowest: |

SECTION 4: DATA DESCRIPTION

DEFINITION

A name in a program may refer to one of many classes of data. For example, it may represent a variable referring to a character string, it may represent a statement label or refer to a pointer or area, etc.

The recognition of data as a particular name is established through declaration of the name.

Those properties that characterize the object represented by the name, and the scope of the name itself, together make up the set of attributes that are to be associated with the name.

Data is established as a name, which holds throughout a certain scope in the program and a set of attributes may be associated with the name by means of a declaration. Declarations must be explicit. Explicit declarations are made through use of the Declare statement, label prefixes and specification in a parameter list. This means that data can be established as a name and can be given a certain set of attributes.

THE DECLARE STATEMENT

The DECLARE statement is a non-executable statement that explicitly specifies attributes which are to be associated with a name.

SYNTAX RULES

DECLARE [level] name [attribute] .... [level] name [attribute]

DECLARE 1 X FLOAT, p FIXED; 2 Y FLOAT, Q FIXED

1. Any number of sets of data may be declared as names in one DECLARE statement.
2. Attributes must follow the names to which they refer.
3. "LEVEL" is a non-zero decimal integer constant. If it is not specified, level "1" is assumed. Only level "1" may be specified for items that are not elements of a structure.
4. A DECLARE statement may have a label prefix, but such does not cause declaration of the name as a label constant.

GENERAL RULES

1. All of the attributes given explicitly for a particular name must be declared together in one DECLARE statement.

PRECEDING PAGE BLANK-NOT FILMED

2. The following attributes may not be specified more than once for the same name: area, based, bit, character, dimension, entry, initial, label, like, offset, precision, and returns.
3. Attributes in different declarations of the same external name must not conflict, nor must they supply information that was not explicit.
4. The attributes declared for external names with the entry or static attributes must be consistent.

DECLARATION OF STRUCTURES

The outermost structure is a major structure and all contained structures are minor structures. Scalar variables or arrays following the elements of major or minor structures are known as base elements. A structure is specified by declaring the major structure name and following it with the names of all contained minor structures and elements.

FACTORING IN DECLARE STATEMENTS

Attributes common to several name declarations can be factored to eliminate repeated specification of the same attribute.

SCOPE OF DECLARATIONS

When a declaration of a data type is made in a block, there is a certain well defined-region of the program over which this declaration is applicable. This region is called the scope of the declaration.

CLASSES OF ATTRIBUTES

DATA ATTRIBUTES

Problem Data Attributes: Attributes for problem data are used to describe arithmetic and string variables. The arithmetic data attributes are: fixed, float, and precision. The string data attributes are: bit, character, and varying.

Program Control Data Attributes: Attributes for program control data specify that the associated name is to be used by the programmer to control the execution of this program. The program control data attributes are: area, entry, event, label, offset, pointer, returns, and task.

ENTRY ATTRIBUTES

The entry attributes identify the name being declared as an entry constant or an entry variable and describe features of the associated entry point. The entry attributes are: entry and return.

SCOPE ATTRIBUTES

The scope attributes are used to specify whether or not a name may be known in another external procedure (program). The scope attributes are: internal and external.

STORAGE CLASS ATTRIBUTES

The storage class attributes are used to specify the type of storage for a data variable. The storage class attributes are: based and static.

ASSIGNMENT OF ATTRIBUTES

ORDERING

Attribute processing takes place in the following order:

1. Defactoring of attributes.
2. Expansion of like attributes.
3. Establishment of explicit declarations.
4. Application of initial attributes derived from subscripted label constants.
5. Resolution of names, including names in attribute.

FORMING A COMPLETE ATTRIBUTE SET

A complete attribute set for a datum, that is, an explicitly declared identifier, a contextually declared identifier, or an implicitly declared identifier, is formed by combining the attributes derived from:

1. Its explicit, contextual, and implicit declarations.
2. The limited system default rules.

associated with a corresponding parameter in the parameter list of the referenced entry point (see "Parameters" below). The number of arguments in the argument list must be equal to the number of parameters in the parameter list of the referenced entry point. In general, any reference to a parameter within the invoked procedure is treated as a reference to the corresponding argument.

The attributes of an argument must match the attributes of its corresponding parameter.

If a programmer wishes to invoke an entry value which requires no arguments, and which for some syntactic reason would otherwise not be invoked, he must write a null argument list, that is ( ).

There are two distinctly different uses for procedures, determined by one of two contexts in which a procedure reference may appear:

1. When a procedure reference follows the keyword CALL, either in a CALL statement or in a CALL option, it is a <u>subroutine reference</u>, and the procedure is invoked as a <u>subroutine procedure</u>, or simply a <u>subroutine</u>.

2. When a procedure reference appears as an operand in an expression, it is a <u>function reference</u>, and the procedure is invoked as a <u>function procedure</u>, or simply a <u>function</u>.

SUBROUTINE REFERENCES

A subroutine reference transfers control to the specified entry point of a subroutine. The subroutine may be terminated normally by execution of a RETURN statement or an END statement; or it may be terminated abnormally by execution of an EXIT statement. On normal termination, control returns to the first executable statement logically following the invoking statement, except when one or more of the TASK, EVENT, or PRIORITY options is specified in the CALL statement (see "Termination of Tasks" in Section 6).

A value is not normally returned by a subroutine, but a value obtained in a subroutine may be made known in the invoking procedure either by assigning the value to a variable known in the invoking procedure, or by assigning the value to a parameter so that the argument obtains the value (in this case a dummy argument must not have been created).

FUNCTION REFERENCES

Function references may invoke programmer-written functions or built-in functions (see "Built-in Functions" in this section).

PRECEDING PAGE BLANK-NOT FILMED

SECTION 5: INVOCATION OF PROCEDURES

A program consists of one or more procedures, each of which may contain other procedures. In general, control is transferred to a procedure through a reference to the name (or one of the names) of the procedure, or to one of its other entry point names.

An entry point is a point in a procedure at which control may be given to the procedure. Every procedure has a primary entry point which is defined by the PROCEDURE statement. A procedure may also have one or more secondary entry points which are defined by ENTRY statements. Each entry point is identified by a name known as the entry name, which is the label prefix of the PROCEDURE or ENTRY statement. A procedure can be executed at any place in a program where an entry name for that procedure is known; this is done by using the entry name in a procedure reference.

PROCEDURE REFERENCES

The procedure invoked by the procedure reference may be an external or an internal procedure.

If it is an internal procedure, the block to which the procedure is internal must be active at the time of invocation of the procedure. See "Dynamic Program Structure".

A procedure reference has the form:

    entry-expression [(argument-list)]

Where "entry-expression" may be:

1. An entry constant (an entry name which is never changed).

2. A scalar entry variable (an entry name which can be redefined, see Section 2).

3. A procedure reference which returns another entry as the value (which results in the procedure of that name being invoked in turn).

Where "argument-list" is:

    argument [,argument] ...

Where "argument" is an expression.

When control is transferred to an invoked procedure, data in the invoking procedure can be made known in the invoked procedure by specifying the data names as arguments in the argument list of the procedure reference. Each argument is

A function reference transfers control to an entry point of a function and activates the function. Activation of the function may be terminated normally by execution of a RETURN statement, or abnormally by execution of a STOP or EXIT statement. The RETURN statement must contain a scalar expression. The value of this scalar expression is then used in the evaluation of the expression containing the function reference.

ENTRY EXPRESSIONS

An entry expression is an entry constant, a scalar entry variable, or a procedure reference which returns an entry value. An entry expression's value is always an entry point. The contexts in which an entry expression may appear are:

1. In a procedure reference.
2. On the right-hand side of an assignment symbol (when assigning a value to an entry variable).
3. As an argument to a parameter (when one of the parameters is expected to be the name of an entry point).
4. In the statements FETCH and RELEASE.

ARGUMENT PASSING

PARAMETERS

When an argument is passed to a procedure, the procedure must have a parameter to accept the argument. A parameter is a name in an invoked procedure that is used to represent an argument passed to that procedure; it may be a scalar, array, or structure name that is unqualified and unsubscripted. An identifier is explicitly declared as a parameter by its appearance in a parameter list.

Each entry point may have a list of parameters. The parameter lists for different entry points of a procedure need not be the same, but only the parameters at the invoked entry point may be used during that execution.

MATCHING ATTRIBUTES OF ARGUMENTS AND PARAMETERS

In procedure invocations, the attributes of arguments must match those of the corresponding parameters and ENTRY attributes if present.

BUILT-IN FUNCTIONS

Besides function references to procedures written by the programmer, a function reference may invoke one of a comprehensive

set of <u>built-in functions</u>.

When a reference is made to a built-in function, any arguments whose attributes do not match the attributes required by that function are flagged as errors. The characteristics of the value returned are determined by the function.

SECTION 6: DYNAMIC PROGRAM STRUCTURE

Dynamic program structure refers to the order in which the various components of a program are executed. The fundamental program element necessary for consideration in this regard is the procedure block. A procedure block is a block of code which has one or more entry points and one or more termination points and over which a certain set of variable names are defined. The procedure is said to be activated when it has been invoked at one of its entries. The procedure is terminated when a 'RETURN' statement is executed, or when another block to which this block is attached is terminated. The procedure block corresponds to the SUBROUTINE facility in FORTRAN, except that it has much more generality. While a given procedure block is activated, it is possible that it may cause activation of another procedure block. In such a case, both blocks are said to be active and the second block is said to be a dynamic descendant of the first. Such activation of subordinate blocks may proceed to any level (limited only by computer resources). If any block causes activation of itself or of any other block of which it is a dynamic descendant, the system is said to be recursive. Recursive activation of blocks is prohibited in the language (unlike PL/I).

During the course of execution of a given procedure block, all the variables declared within the procedure block are available for computation as well as the variables which have been pased to the procedure as parameters in the activation

statement (CALL statement). The latter, however, belong to the block from which the procedure block descended. In addition, any variables which have been declared to be EXTERNAL in the compilation of the procedure block are available for computation.

Storage for variables declared within a procedure block may be either of STATIC storage class or of the BASED storage class. STATIC storage class implies that storage for the variables is allocated with the allocation for the code of the procedure itself, i. e., at "LOAD" time. Allocation for BASED variables is completely under the control of the programmer. No allocation is performed for this class of variables until the execution of an 'ALLOCATE' statement naming the particular BASED variable. Associated with each BASED variable must be a pointer variable. The allocation for the pointer variable itself may be BASED or it may be STATIC.

In addition to ordinary subroutines, the language supplies the ability to specify that a procedure should be executed independently of the calling routine. This ability implies that if two processors are available for execution, one processor may begin executing the procedure while the other processor continues executing the calling program. The usefulness of this concept comes not in its application to two processor systems, but to multi-task systems where several functions must be performed simultaneously with different time requirements. Such an independently executed procedure is called a parallel task and it is indicated by the use of the 'TASK' option in the calling

sequence for the procedure. Each such independent task can itself create other tasks to any level. The task so created is said to be "attached" to the task creating it although in general no return of control from the created task to the creating task ever occurs. The concept of attachment is necessary for the purposes of determining ownership of allocated storage.

In order to accommodate the concepts necessary for a process control computer system, we view the entire operating system as the '<u>MAIN</u>' program of the language being executed by the system. Note that this concept is completely different from that of PL/I wherein the operating system is a set of support facilities which enable certain features of the language to be implemented. In the view of Pl/I, if a proper computer were to be constructed, one which had as its hardware instructions the primitive elements of Pl/I, no operating system would be necessary -- the machine would execute PL/I programs directly. Our view of the total system is somewhat different. We accept the computer as built, and demand of the language that it be able to provide the features necessary to create an operating system. In this way we can describe all the facilities available within the operating system in the language itself as well as the actions of all the peripherals. Each application program in this view becomes a subordinate task which is attached to the operating system.

The communication between a subordinate procedure and the procedure which called it is by means of shared program storage. This shared storage may be in an argument list passed to the called routine, through EXTERNAL variables, or through variable declarations common to the two routines. The last type of shared storage occurs if one routine is contained within another at compilation. Flow of control proceeds from one routine to another by means of 'CALL' and 'RETURN' statements. Within a parallel task, however, there is no return of control to the calling task. Each parallel task executes independently until completion. Flow of control enters the parallel task by means of a 'CALL' statement as before except that this does not suspend the execution of the calling task. Communication is similar to that for fully dependent subroutines except that the only variables available to the parallel task are those of the task to which it is attached. Normally the called task is attached to the task which called it, but this may be altered by means of the 'ATTACH' option in the calling sequence. By specifying that a newly created task should be attached to the 'SYSTEM', one can create a completely independent task.

The handling of the arguments within the execution of such a call must be noted. Normally the arguments belong to the calling task; the called task can refer to them as necessary. This is often referred to as a call by name. When the ATTACH option is specified, however, the values of the parameters are copies, and the copy of the values are attached to the task named in the ATTACH option for the called task.

In addition to these methods of data communication, a facility is provided so that tasks may be synchronized. This facility is based on an event structure which the language assumes. An event is a data type subject to all the rules concerning data names. Each single event variable is concerned with the occurrence of some "happening" within the computer. Associated with each event variable is a logical value, true or false, which measures whether or not the event has occurred, and an integer value which measures the nature of the occurrence. The precise interpretation of this integer is undefined within the language and is at the option of a given implementation. If the event variable is denoted E, then the logical value can be accessed by the pseudo-variable and built-in function COMPLETION (E), and the integer value can be ascessed by the pseudo-variable and built-in function STATUS (E). An event may be associated with the completion of a task by specifying an event name in the 'EVENT' option in the calling sequence for the task. The names event becomes active and the completion and status values described above are defined. The calling task may interrogate these values to ascertain the current state of the subordinate task. In addition, the calling routine may suspend itself until the called task completes. It does this by means of the WAIT statement specifying the event associated with the subordinate task as an argument. This statement causes the program executing it to be suspended until the event names has occurred. At that point the program resumes at the statement following the WAIT statement.

Events may also be associated with the completion of input-output operations; see the section on I/O for more details.

In general, event variables follow all the rules for definition of names as do other variables. The allocation procedures appropriate to STATIC and BASED storage apply to EVENT variables as well, inclduing the requirements on pointer variables implicit in BASED storage.

Within the MAIN program (i.e., the operating system), there is another data type available, the DEVICE variable. One device variable must be declared for each physical device available on the machine. Associated with each device variable is a set of values appropriate to the device. Each of these values may be accessed through a suitable built-in function and may be set by a suitable pseudo-variable. For example, most physical devices will have some sort of status flag associated with them which is used to indicate device status. This value (logical true or false) may be accessed by the FLAG built-in function. That is, if D is a device variable, then FLAG (D) is the logical value of the device flag. The flag may be cleared by the use of the FLAG pseudo-variable, e. g., FLAG (D) = OB. If the device requires an input or output buffer area for operation, the current value of the pointer to the buffer area is available through the use of the BUFFER built-in function. That is, BUFFER (D) is a pointer value to the current buffer area and this value may be set by the BUFFER pseudo-variable, e. g., BUFFER (D) = P.

Other values may similarly be associated with a device as appropriate. Besides the static values which can be associated with a physical device, a device state may also undergo transitions which may be significant. Such a transition may be for example, the receipt of a character from a teletype. These transitions can be classified as events under the broad definition given above, and we need a formal way to create an event from a device transition. This is provided by the OCCUR built-in function. The statement

```
E = OCCUR (D)
```

will result in the event E taking place on the next transition of the device D. In this way the sequence of code

```
E = OCCUR (D)
WAIT (E)
```

or even more concisely,

```
WAIT (OCCUR (D))
```

will result in the program being suspended until the next occurrence of a transition on device D. The application programs written as tasks attached to the operating system, have available to them all the information about the devices they need. Through the event structure, they can make computations conditional upon the occurrence of device transitions.

Within the operating system itself, that is within the MAIN program, there is an additional statement available to the language to handle the devices. This is the ON statement

which describes the actions to be taken when the device causes an asynchronous interrupt. The form of the statement is

ON (device name) statement

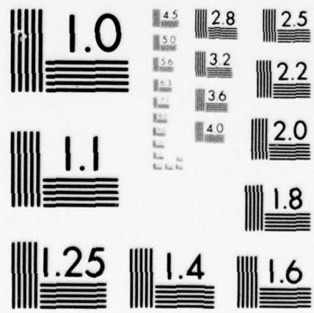
where (device name) can be any variable which has been declared to be a device variable. The statement is a language statement which is to be executed when the named device causes an interrupt. This statement can be compound, that is it can be a DO group. The use of this statement is reserved to the MAIN program to provide some discipline in device interrupt responses. Depending on the structure of the operating system, the application program may cause a routine to be activated when a device interrupt occurs by means of the event structure and the OCCUR built-in function, or it may cause a routine to be directly executed by supplying its name to the operating system.

AD-A036 475 PURDUE UNIV LAFAYETTE IND PURDUE LAB FOR APPLIED IND--ETC F/G 9/2
SIGNIFICANT ACCOMPLISHMENTS AND DOCUMENTATION OF THE INTERNATIO--ETC(U)
JAN 77 N00014-76-C-0732

UNCLASSIFIED NL

2 OF 3
AD
A036475

03647



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

SECTION 7: INPUT - OUTPUT

The general area of input/output operations is one of the largest unresolved problem areas of this language design effort. The I/O operations provided for in PL/1 are monumentally complex when considered in their totality, and unsuited for our requirements. Specifically, the approach to file systems is much too complicated and the concept of process I/O is totally absent.

Our work on I/O has been divided into three topics:

1. Use of bulk memory devices.

2. Device Dependent I/O (e.g. Process I/O)

3. Device Independent I/O

The emphasis of the work which has been done is to provide a lucid, simple set of facilities to accomplish I/O operations in as efficient a manner as possible. For example, the operation of format conversion has been purposely separated from I/O operations to eliminate unnecessary repetitive operations.

The material to follow is in the nature of proposals rather than final specifications. Development of a concrete syntax and use of that syntax will probably lead to significant changes.

Use of Bulk Memory

A typical facility for the use of bulk memory is a full file system. Such a system may be found in PL/1. If we consider a collection of data external to a program to be a data set, a file system provides a mechanism for insuring the life of a data set independent of the life of any process which creates or utilizes it. The parameters describing the data set are known throughout the system until the existence of the data set is explicitly terminated by some process.

Associated with such a system are a significant number of data structures describing the files known to the system, their current state, conditions for using them, etc. File systems such as this require a large package of supporting software and incur significant overhead penalties during execution.

More appropriate to computer control is a system which provides for dynamic allocation of bulk store and easy manipulation of data and program found there. (This is consistent with the elimination of AUTOMATIC dynamic memory allocation

and the subsequent use of only based storage for dynamic storage allocation).

Proposed are:

1. The attribute BULK BASED which refers to dynamically allocated bulk storage.

2. The ALLOCATE statement used to cause allocation of the bulk store.

3. a) FETCH and RELEASE statements for program files.

   b) A presently undetermined statement which will effect copying of data files between bulk and core.

This approach provides a simple, consistent means for handling dynamic allocation of bulk memory devices, and could easily be used to implement a complete file system. Its usage conventions will correspond to those for BASED variables.

PROCESS I/O

The information required to be passed in a process I/O call may be categorized as follows:

1. Information which designates the physical device from which the data is to be obtained. (e.g. Analog Input Scanner or Digital Output Module).

2. The source within the physical device of the process variable (on input) or specific output path for output. (e. g. analog input point (group) number or digital output point designation).

3. The destination (source) of the input (output) data.

4. Conversion parameters for I/O device.

5. Information relating to the flow of control desired for the operation (i. e., should program be roadblocked until the operation is complete or continue computation).

All process I/O information falls into these categories. It will be handled as follows:

1. A reserved word <device name>, designating the device.

2. A structure name <device descriptor structure> to be passed indicating the required data path; or

a variable name pointing to the data required.

3.

4. Data structure or variable names as appropriate as arguments.

5. (Optional) An event variable. If present all processes attached to this event variable will be activated upon completion of the I/O activity. In addition, any code following the I/O operation will be examined immediately. A syntax for process I/O statements is:

{INPUT / OUTPUT} <device name> (device descriptor structure)

<argument structure name> [, EVENT (event name)]

EXAMPLE

INPUT SCANNER (5,70,20), A (1), TEMP 2

External I/O

External I/O operations will be handled as specific character-string transmissions. An output operation statement would have the syntax:

WRITE <Logical Device Name>, <Character String Expression>

The number of characters transmitted in the string will be determined by the length of the Character String Expression.

Format conversions etc. may be handled by built-in functions. The string manipulation features of the language will make this a natural way to handle this.

SECTION 8: STATEMENTS

This section includes an alphabetical list of all the statements in the language with a brief description of the function of each.

ALLOCATE: Causes storage to be allocated for specified based variables.

ARM: Allows a hardware interrupt to occur so that it may be recognized and remembered.

Assignment: Evaluates an expression and assigns its value to one or more target variables. The target variables must be scalar, and may be indicated by pseudo-variables. No implicit conversions between data types are allowed.

CALL: Invokes a procedure and causes control to be transferred to a specified entry point of the procedure. May include an argument list and TASK, EVENT, PRIORITY, and ATTACH options.

COPY: Assigns the data from one data structure to the elements of another data structure, where the types of the data elements and their arrangements in the structure have been explicitly specified in previous declaration statements. Transformations from one data type to another will be permissible where this is meaningful.

DECLARE: A non-executable statement that explicitly specifies attributes which are to be associated with a name.

DELAY: Causes execution of the controlling task to be suspended for a specified period of time.

DISABLE: Prevents the processing of an interrupt. This is the inverse of the ENABLE statement.

DISARM: Causes an interrupt signal to be completely ignored. This is the inverse of the ARM statement.

DO: Delimits the start of a DO group and may specify repetitive execution of the statements within the group. May include a specification of a set of values to be taken on by a variable or pseudo-variable during successive executions of the group, and may include a WHILE clause specifying the condition for continued repetition.

PRECEDING PAGE BLANK-NOT FILMED

SECTION 8: STATEMENTS

This section includes an alphabetical list of all the statements in the language with a brief description of the function of each.

ALLOCATE: Causes storage to be allocated for specified based variables.

ARM: Allows a hardware interrupt to occur so that it may be recognized and remembered.

Assignment: Evaluates an expression and assigns its value to one or more target variables. The target variables must be scalar, and may be indicated by pseudo-variables. No implicit conversions between data types are allowed.

CALL: Invokes a procedure and causes control to be transferred to a specified entry point of the procedure. May include an argument list and TASK, EVENT, PRIORITY, and ATTACH options.

COPY: Assigns the data from one data structure to the elements of another data structure, where the types of the data elements and their arrangements in the structure have been explicitly specified in previous declaration statements. Transformations from one data type to another will be permissible where this is meaningful.

DECLARE: A non-executable statement that explicitly specifies attributes which are to be associated with a name.

DELAY: Causes execution of the controlling task to be suspended for a specified period of time.

DISABLE: Prevents the processing of an interrupt. This is the inverse of the ENABLE statement.

DISARM: Causes an interrupt signal to be completely ignored. This is the inverse of the ARM statement.

DO: Delimits the start of a DO group and may specify repetitive execution of the statements within the group. May include a specification of a set of values to be taken on by a variable or pseudo-variable during successive executions of the group, and may include a WHILE clause specifying the condition for continued repetition.

PRECEDING PAGE BLANK-NOT FILMED

ENABLE:	Allows the processing of an established interrupt.
END:	Terminates blocks and DO groups.
EXIT:	Causes immediate termination of the task that contains the statement and all tasks attached to this task.
FETCH:	Causes the procedures specified to be fetched into internal storage for subsequent execution, unless suitable copies of the procedures already exist in internal storage. May include an EVENT option.
FREE:	Causes the storage allocated for specified based variables to be freed.
GOTO:	Causes control for a task to be transferred to the specified statement within the same block.
IF:	Specifies evaluation of a bit string expression and a consequent flow of control dependent upon the value of the bit string.
INPUT:	Causes data to be read in from the specified device to the specified variables.
NULL:	No operation.
ON	Specifies the action to be taken when an interrupt occurs for the named device.
OUTPUT:	Transmits data from the specified variables to the specified device.
PROCEDURE:	Identifies a portion of the program text as a procedure, defines the primary entry point to the procedure, and specifies the parameters for the primary entry point.
RELEASE:	Specifies that internal storage containing specified procedures may be freed for other purposes.
RETURN:	Terminates execution of the procedure that contains the RETURN statement. If the procedure has not been invoked as a task, the RETURN st atement returns control to the invoking procedure. It may also return a value.

SIGNAL:	Causes the occurrence of the named event.
STOP:	Causes immediate termination of the major task and all subtasks.
WAIT:	Retains control within the WAIT statement until the specified events have occurred.

SUMMARY OF PREFERABLE FEATURES OF PROCEDURAL LANGUAGE FOR INDUSTRIAL USAGE

H. Kuwahara

Omika Works of Hitachi Ltd.

It is very important, we believe, to make clear the preferable or necessary features required in process language in the course of development of its specification.

Attached table shows the preliminary results of our efforts. In order to make the work easy and based on the prospect that the preferable language will be of higher level language than FORTRAN, we described necessary features desirable to be added to standard FORTRAN.

Some of these features are well covered in PL/1 or ALGOL and some others are not.

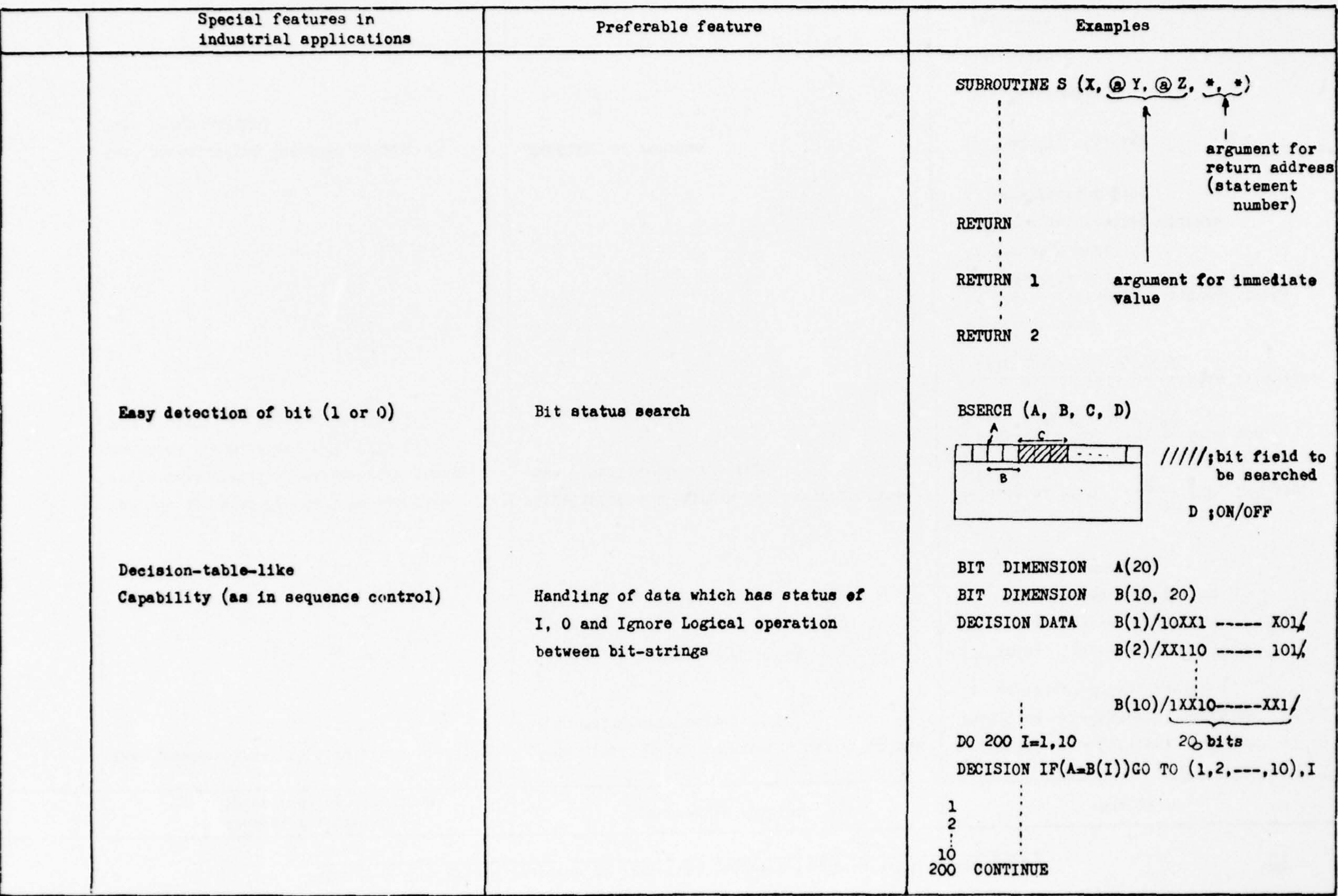
We think we should make our efforts to establish mutually acceptable preferable features of process language, which will be useful and indispensable when we want to make standards of several levels or when we need to give up syntax or semantics which are fancy but make object programs longer.

PRECEDING PAGE BLANK-NOT FILMED

Summary of Preferable Features of Procedual Language for Industrial Usage (1/5)

	Special features in industrial applications	Preferable feature	Examples
Data	Many FIXED and less FLOAT arithmetic data	FIXED constant/variable of one/two word length	INTEGER *2
		Options of expanding inplicit declaration of data (not limited to I~N)	IMPLICIT INTEGER (A-D, X-Z)
	Handling of many one bit or several bit status information within a limited amount of memory	Constant/Variable data of more than one bits	BIT DIMENSION A(5) BIT DATA A(1)/1101
		Easy expression of bit-string data	1A(2), 2B, 3C: BIT(6), 3D: BIT (10) 2E, 3F: BIT(8), 3G: BIT (8)
		Three dimensional data structure (capability of expressing data of more than one bits)	
		Capability of bit-string operation	C(1) = F(2) & G(2) G(1) = F(2) 1 G(2) F(2) = F(2) % G(2) D(1) = ¬ D(2)
	Very frequent and sophistidated data transfer between main and bulk memory	Introduction of virtual memory philosophy (handling of data transfer without any special data-structual consideration to different memory devices)	The same data structure for the data on bulk memory BULK DIMENSION A(100), C(100) DIMENSION B(100) ⋮ A(I) = B(I)*C(I)
Programm-ing	Efficient and easy generation of problem oriented macro statements	Statement number could be an argument of CALL statement (example; for return address indicator)	Y=1000 ⋮ CALL S(X, @ Y, @ 2000, & 100, & 200) (@ Y, @ 2000: imediate value; & 100, & 200: statement number)
	Easy linkage with sub-routine coded by assembler language	Imediate value(constant/variable) could be an argument of CALL statement	

Summary of Preferable Features of Procedual Language for Industrial Usage (2/5)

	Special features in industrial applications	Preferable feature	Examples
			SUBROUTINE S (X, @ Y, @ Z, *, *) RETURN RETURN 1 RETURN 2 @ Y, @ Z: argument for immediate value *, *: argument for return address (statement number)
	Easy detection of bit (1 or 0)	**Bit status search**	BSERCH (A, B, C, D)  /////;bit field to be searched D ;ON/OFF
	Decision-table-like **Capability (as in sequence control)**	**Handling of data which has status of I, 0 and Ignore Logical operation between bit-strings**	BIT DIMENSION A(20) BIT DIMENSION B(10, 20) DECISION DATA B(1)/10XX1 ----- X01/ B(2)/XX110 ----- 101/ B(10)/1XX10-----XX1/ (20 bits) DO 200 I=1,10 DECISION IF(A=B(I))GO TO (1,2,---,10),I 1 2 10 200 CONTINUE

Summary of Preferable Features of Procedual Language for Industrial Usage (3/5)

	Special features in industrial applications	Preferable feature	Examples
	Easy accessibility to hardware capability	Accessibility to hardware registers (other than arithmatic register)	A=IREG(i) ---- internal register EREG(i)=B ---- external register i; register address number
		Control of hardware registers	ISIFTA(i_1, i_2) ---- arithmetic ISIFTL(i_1, i_2) ---- logical i_1; register address number i_2; shifting number
	Various types of I/O devices and frequent communication between main memory and them (Drum, Disc, M/T, C/R, L/P, PTR/P, CRT, KB, Process I/O etc.)	Simplified and uniform I/O control statement (including formating)	Sequential File ------- L/P, C/R etc. READ(u,f,e)K Random Sequential File-- Drum etc. READ RSFILE (u,a,f,e)K Random Random File ----- Data file etc. READ RRFILE (u,a,f,e)K u: device name a: address(for RS)and address table name (for RR) f: format number e: error information address k: input/output list
	Data accumulation (sampled analog data accumulation)	Handling of residue	i_4= MOD (i_1, i_2, i_3) i_1; dividend i_2; divisor i_3; residue i_4; answer

Summary of Preferable Features of Procedual Language for Industrial Usage (4/5)

	Special features in industrial applications	Preferable feature	Examples
Tasking	Timing control between tasks and hardware resources/external events	Interrupt handling	INHON INHOFF
	Multi-programming	Separation of relatively slow I/O control program as an independent task (data buffering)	BUFW (A, F) K ----- buffer write QUEUE n (A: buffer name F: format number K: output list n: output control task using buffer A) BUFR (B, F)K ----- buffer read (B: buffer name F: format number K: input list)
	Frequent communication between tasks	Data communication (1) through main memory (2) through auxiliary memory Synchronization of tasks (1) through task management system in Operating System (2) through data management system in Operating System	GLOBAL a_1, a_2, ----- , an BULK GLOVAL e_1, e2, -----, en BIT GLOVAL v_1, v_2, -----, vn ai, ei, vi are DIMENSION data such as in PL/1

Summary of Preferable Features of Procedual Language for Industrial Usage (5/5)

	Special features in industrial applications	Preferable feature	Examples
Debugging	Easy handling of statement/sub-routine which are used only in debugging phase and should be eliminated from object program	Switching of effectiveness of statement/ sub-routine between off-line debugging mode and object program generation mode	Effectiveness control of statement comment, data etc. RENAME (A,AI), (B,BI) which replace sub-routine A by AI and B by BI
	Tracing of program flow	Type out of specified variable value on specified condition	DEBUG VALUES n_1, n_2 (IF(e)a_1,a_2,---- an) n_1, n_2;difine trace range ai ; variable to be typed out when it is on the left side of equation
		Type out of destination address when jumping instructions are executed	DEBUG JUMP n_1, n_2 IF(e)
		Range over check of subscript of subscripted data	DEBUG SUBSCRIPT a_1, a_2, ---- ,an ai; array name to be checked
		tracer etc.	

SECTION II

WORKING AND POSITION PAPERS

TASKING PROPOSALS FOR THE LONG TERM PROCEDURAL LANGUAGE

Tasking, or the management of parallel processes, is one of the most important areas of consideration in the design of the Long Term Procedural Language because of the importance of these features to operation in the industrial environment. As such it has been a very popular topic of discussion and writing in the LTPL Committee and its branches. The following documents are a nonexhaustive sampling of some of the most important of these findings. They are contained in the Minutes of the International Purdue Workshop on Industrial Computer Systems and its predecessor, The Purdue Workshop for Standardization of Industrial Computer Languages as follows:

1. "Position Paper-Tasking, Interrupt Handling, and the Synchronization of Parallel Computational Processes", Minutes, Fifth Purdue Workshop on Standardization of Industrial Computer Languages, pp. 225-231, by H. Pike.

2. "Task Interactions", Ibid, pp. 235-246, by P. Lindes.

3. "Tasking", Minutes, Sixth Workshop on Standardization of Industrial Computer Languages, pp. 196-202, by P. Rieder; pp. 204-231 by J.G.P. Barnes, R. J. Long, and D. N. Shorter.

4. "Joint Tasking Proposals for an LTPL", Minutes, 1974 Spring Regional Meetings, International Purdue Workshop on Industrial Computer Languages, Document IV-II-VI, pp. 371-390, by D. N. Shorter and K. H. Timmersfeld.

The reader is also urged to review Document 3 of Table I of the Introduction of this Part for the comparison of the tasking features of each of the languages considered there.

-LTPLC -

POSITION PAPER

TASKING, INTERRUPT HANDLING, AND THE SYNCHRONIZATION OF PARALLEL COMPUTATION PROCESSES

H. PIKE

APRIL 1971

INTRODUCTION

Perhaps the most unique programming requirements associated with the implementation of software for real-time industrial systems are reflected in the need for facilities in the programming language for interrupt handling, synchronization of parallel computational processes, and tasking. Although there exist in PL/I features addressed to these problem areas, they are not adequate for our requirements, as we will see below. The purpose of this paper is to unify our thinking on these topics by developing a clear definition of the problems which we are addressing, examining current work to date, and proposing a solution. The solution is an attempt to further define the approach proposed by Neal Laurance. Credit for the idea should be given to Neal, blame for "what has been done to it" can be attributed to the author!

DEFINITIONS

Segment: A sequence of executable statements

Computational Process: A flow of control executing a segment

Task: A virtual processor or flow of control capable of executing statements or segments.

Parallel Computational Processes: Computational processes which take place simultaneously.

Event: An asychronous occurrence external to a computational process, but of interest to the computational process.

Main Task: In an industrial computer system, the task whose responsibility includes supervision of the execution of all other tasks. (e.g. an "operating system").

Subordinate Task: A task which is not a main task.

Interrupt: An event external to a main task and its subordinate tasks.

Synchronization of Parallel Computation Processes: Controlling the execution of parallel computational processes so as to maintain some desired sequential relationship between events caused by the processes.

Status Variable (or state variable): A variable which assumes a state from a set of possible states, and whose state may charge upon the occurrence of an event.

PROBLEM STATEMENT

It is desired to develop a set of facilities for the language, including basic data types, operations, system primitives, etc. to fulfill the functional requirements for industrial computer systems as outlined below. This is to be done in a manner which maximizes conformity with PL/I.

FUNCTIONAL REQUIREMENTS

In order to define the functional requirements (or, more properly, functions which must be directly implementable in the language), we must first take a system overview of what is to be accomplished.

The approach taken to this problem in PL/I is to define a "PL/I machine"; that is a computer whose hardware executes PL/I programs directly. Thus considerations of an operating system disappear.

It is our view that we must be somewhat more precise than this. We take as our system model a simplified version of the PL/I machine executing a main task and possibly subordinate tasks. The capabilities of this processor will be required to be such that a modern conventional processor with some minimum

kernel software could provide this restricted version of our PL/I machine. We desire, as far as possible, to keep this to a proper subset of the PL/I machine for obvious reasons.

Subordinate tasks will view their environment as a virtual computer which is an extension of the PL/I machine seen by the main task, because functions in the main task will be available to them. Let us call these two machines (for the want of better names): the PL/I Subset Machine, and the LTPL machine.

We are now ready to discuss the functional requirements for the language. It will be clear that some facilities will not be available to the implementer of the main task because they do not exist on his "PL/I Subset Machine"; they will, from his point of view, be system functions he must provide for the subordinate tasks.

From a linguistic point of view it is not necessary to restrict language facilities available to the main task from subordinate tasks. In practice, two factors will cause this to be done:

1) System security - the scope of capability of subordinate tasks will be restricted from operations which could impair system security.

2) Real time performance - certain operations will imply or require real-time performance. This can only be assured to the main task, so these facilities will be restricted to the main task.

With the above in mind, we will specify desired functional capabilities in terms of requirements on the PL/I Subset Machine and the LTPL Machine.

FUNCTIONAL REQUIREMENTS: PL/I SUBSET MACHINE

INTERRUPTS:

Ability to initiate a computational process immediately upon the occurrence of an interrupt.

- Ability to insure that a computational process is completed before any other computational process is initiated.

- Ability to determine if some interrupt has occurred during some past time interval.

SUB-TASK INITIATION -

Not explicit at this level

SYNCHRONIZATION -

Semaphores or equivalent

FUNCTIONAL REQUIREMENTS: LTPL MACHINE

INTERRUPTS:

- Ability to cause the initiation of a computational process on the occurrence of an interrupt by indicating a segment which is to be executed if the interrupt occurs.

- Ability to insure the "prompt execution" of such a process.

- Ability to mask the connection between a segment and an interrupt with two possible actions on unmasking:

  a) ignore any interrupts which occurred while masked

  b) execute upon unmasking if any interrupts occurred while masked

TASKING

- establish an external procedure as a subordinate task with possible parameters:

  a) priority

  b) time

  c) time interval

  d) procedure name

- delay a task
- suspend a task
- un-suspend a task
- exterminate a task
- check status of a task
- re-assign priority
- semaphores for synchronization

PRIOR WORK

Work of direct interest to this proposal comes from three sources: the current version of PL/I, the proposed PL/I standard, and Laurance's LTPL work.

Since Laurance's work is readily available to the committee, I will only attempt to describe the current trend of the PL/I standard. The best indication of current X3JI thinking is development proposal X3JI.2/80 by Freiburghouse "A Proposed Tasking Facility for PL/I". This proposal is to completely replace the current tasking facilities in PL/I.

Freiburghouse has proposed tasks and subtasks as in the current language, with a major task being created with the invocation of the first procedure of the PL/I program, and subtask creation by execution of a call statement with the task option. Task termination occurs upon execution of a return statement in the major task, or a stop-statement anywhere.

Synchronization is to be accomplished through a two state lock variable, which resides in constant storage and cannot receive new values through assignment or input operations. The usual semaphore statements are defined on lock variables. Also included in the proposal is the wait statement, although its function can be performed with lock statement, i.e.:

WAIT (x) is equivalent to LOCK (x); UNLOCK (x), assuming x is initially locked and the transition from locked to unlocked state is the event of interest.

The proposal includes the use of lock variables for synchronization of tasks and events by association of the lock variable with an event. Event variables are eliminated.

PROPOSAL

The basic assumptions of this proposal include the functional requirements above and that a lock variable such as that proposed by Freiburghouse will be included in PL/I. It is also assumed that we cannot use event variables as they are now found in PL/I.

1) Eliminate event variables. The confusion between the occurrence of an event and the value (?) of an event variable is immense.

2) Introduce the status variable (or state variable) as a generalization of the lock variable. (retain lock variables as a special two-state status variable).

3) Provide semaphore operations on lock variables as per proposal X3JI.2/80.

4) Provide the wait statement, defined to cause execution of a segment upon any state transition of the status variable which is its argument.

5) Restrict the use of the wait statement to subordinate tasks.

6) Provide the on-statement for direct linkage of a computational process to an event external to the major task (i.e. a hardware interrupt).

7) Restrict use of the on-statement to the major task.

8) Provide a set of tasking primitives as built-in functions (procedures) to satisfy the tasking functional requirements.

9) Restrict the tasking primitives to subordinate tasks.

10) Provide a set of event variable primitives to fulfill the interrupt handling functional requirements on the LTPL machine. (e.g. MASK, UNMASK etc.)

11) Restrict these to subordinate tasks.

SYNTAX AND SEMANTICS (INCOMPLETE!)

The syntax and semantics will be divided into that for the PL/1 subset machine and that for the LTPL machine. This syntax proposal should be viewed as preliminary, particularly with respect to the LTPL machine primitives.

PL/I MACHINE

The on-statement will be used to associate a segment with the potential occurrence of an external event (interrupt): on-statement::= ON (interrupt-identifier), [PRIVILEGED] ;

Semantics:

Upon occurrence of the interrupt the procedure headed by the on-statement is entered. Upon completion of the procedure, control returns to the interrupted procedure. Switching on-blocks for one interrupt is accomplished by using dummy interrupt identifiers and assignment statements. Inclusion of the privileged option will prevent the recognition of other interrupts until the current on-block is complete.

The status variable is a variable which assumes a value from a set of possible values declared for it. Assignment statements can be used to change its values. Indivisible semaphore operations will be provided.

LTPL MACHINE

The suntax for the LTPL machine will include:

1) The wait statement causing block activation upon the change of state of a status variable.

2) Tasking primitives as defined above.

3) Interrupt masking primitives as discussed above.

TASK INTERACTIONS

LONG TERM PROCEDURAL LANGUAGE COMMITTEE

WORKING PAPER

P. LINDES

APRIL 29, 1971

INTRODUCTION

The purpose of this working paper is to summarize a number of proposals on tasking that have come to the attention of the LTPLC. This summary will be used by the Committee as a basis from which to develop a complete position paper on the subject. First, the functional requirements for tasking operations will be stated as presented in Herb Pike's Draft Position Paper (Reference 1). Then the tasking operations included in each proposal will be compared with these functional requirements.

FUNCTIONAL REQUIREMENTS

The functional requirements for tasking operations available to the applications programmer are (Ref. 1):

1. Establish an external procedure as a subordinate task with possible parameters:
   a. priority
   b. time
   c. time interval
   d. procedure name
2. Delay execution of a task for a specified time period.
3. Suspend the execution of a task.
4. Unsuspend (re-activate) a task.
5. Exterminate a task
6. Check status.
7. Reassign priority.
8. Semaphores for synchronization.

PRECEDING PAGE BLANK-NOT FILMED

The following paragraphs summarize how each of several proposed languages meets these functional requirements.

PL/I (Refs. 2 and 3)

1. A task is created by the execution of a CALL statement containing one or more of: TASK option, EVENT option, or PRIORITY option. A task is a "dynamic descendant" of and depends for its existence on the task that created it. The TASK option can give a name to the task.

   a. The priority of a task can be specified by the PRIORITY option in the CALL statement that created it or by assigning a value to the PRIORITY pseudo-variable for the task name before the task is created. If not specified, the priority is the same as that of the creating task.

   b,c. No time or time interval can be assigned to a task.

   d. When a task is created, it executes the procedure named in the CALL statement that created it.

2. DELAY(n) suspends the task which executes it for n milliseconds.

3. A task can be suspended by execution of a WAIT statement, which specifies an event variable. This event variable can be associated with a subordinate task if it was specified in the EVENT option of the CALL statement that created the task, or it can be associated with an I/O operation by using the EVENT option on an I/O statement.

4. A task that is waiting for an event is unsuspended by the "completion" of that event. This can only occur if the task or I/O operation associated with the event variable is completed (i.e., the associated task is terminated).

5. A task is terminated if: the task executes a RETURN or END statement in the procedure invoked by the creation of the task; the task executes an EXIT statement; any task executes a STOP statement; or the task which created this one is terminated.

6. The "completion" of an event can be tested with the completion built-in function for the associated event variable. After an event has been completed, the manner in which it completed can be tested with the STATUS built-in function. The priority of a task can be examined with the PRIORITY built-in function.

7. The priority of a task can be assigned before its creation with the PRIORITY pseudo-variable, or at its creation with the PRIORITY option. The priority can not be reassigned while the task exists.

8. There is no means for synchronizing tasks other than the WAIT statement, which waits for the completion of the corresponding task. Preventing unpredictable or undefined results caused by simultaneous access to shared data or files is left entirely to the ingenuity of the programmer, except for the special case of locked records in EXCLUSIVE DIRECT UPDATE files.

This PL/I approach has several distinct deficiencies:

i. There is no way to create a task that will exist permanently independent of the task which created it.

ii. There is no way to specify the time at which a task is to be created.

iii. A suspended task cannot be unsuspended except by the termination of the task it is waiting for. Thus no intermediate synchronization is possible.

iv. A STOP statement terminates all tasks.

v. The only status information that can be checked while a task is executing is the PRIORITY and COMPLETION built-in functions.

vi. The priority of a task cannot be changed while it is executing.

vii. One task cannot terminate another (except by terminating itself).

viii. There is no semaphore mechanism for synchronizing access to shared resources.

Proposed New Tasking Facility for PL/I by R. Freiburghouse*(Ref.

1. A task is created with a TASK option on a CALL statement as in PL/I. Instead of an EVENT option, a lock can be specified on the TASK option. The lock is locked when the task is created and unlocked when the task terminates.

   a. There is no provision for priority.

   b,c. Time or time interval cannot be specified.

   d. A procedure name is given by the CALL statement as in PL/I.

2. The same DELAY statement as described above for PL/I, except that the units of time in which the delay is specified are implementation defined.

3,4. Suspension and unsuspension of tasks can be done with LOCK, UNLOCK, and WAIT statements. See 8 below.

5. Tasks are terminated in the same way as in PL/I.

6. There is no status checking except for that provided by LOCK and WAIT statements.

7. Priority cannot be reassigned since no priority exists.

8. Freiburghouse's paper proposes a "lock," which has two states: "locked" or "unlocked." A lock can be associated with a task when the task is created. The LOCK statement specifies a lock and an optional ELSE group. When this statement is executed the following happens:

   If the lock was not already locked, it is locked and the task continues to execute. This operation is indivisible.

   If the lock was locked and there is an ELSE group, the ELSE group is executed.

   Otherwise the task executing the LOCK statement is put in a queue to wait for the lock to become unlocked.

---

*The paper discussed here is a proposal made to X3J12 for modifications to PL/I. The part of the language not specifically discussed in this paper is the same as PL/I.

The UNLOCK statement unlocks the lock. If one or more tasks are waiting in the queue, one is taken out of the queue and allowed to continue execution, and the lock is immediately locked again. This entire operation is indivisible. The language does not define which task is taken from the queue.

The WAIT statement is equivalent to a LOCK statement immediately followed by an UNLOCK statement for the same lock.

The differences between the Freiburghouse proposal and PL/I can be summarized as follows:

i. It contains no provision for priority.

ii. It does not have any equivalent of the STATUS built-in function.

iii. It has all the deficiencies of the PL/I approach <u>except</u> that it has a reasonable synchronization mechanism with the locks.

<u>LTPL Report to Fourth Workshop</u> (Ref. 5)

The tasking facilities in this LTPL report are the same as in PL/I with the addition of the ATTACH option to the CALL statement. The ATTACH option specifies a task to which the new task being created is attached. The special task name "SYSTEM" can be used to attach the new task to the operating system. This proposal has the same advantages and disadvantages of the PL/I approach except that the ATTACH option allows a task to be created which can exist independently of the task which created it.

RTL (Ref. 6)

In RTL the term "thread" is used to denote an entity which corresponds to a "task" in PL/I. Threads are controlled by "supervisor calls."

1. A thread is created with a MAKETHREAD supervisor call, but does not start executing until started by a START call.

   a. Priority is specified by the MAKETHREAD call.

   b,c. No time or time interval can be specified.

   d. A procedure name cannot be specified directly. The instructions to be executed by the thread are determined by information in a "save area" which is specified when the thread is created.

2. There is a DELAY call equivalent to the DELAY statement in PL/I.

3. A thread can suspend itself with a WAIT call which specifies an event.

4. A thread is unsuspended if another thread executes a STIMULATE call which specifies the event for which the first thread was waiting.

5. A thread can "stop" another thread with a STOP call. It can stop itself with a STOPME call. A stopped thread can be restarted with a START call from another thread. A thread can completely destroy itself with a KILLME call.

6. One thread can check the status of another by WAITing for an event which the other thread STIMULATEs, or through shared data.

7. The priority of any thread can be reassigned with a CHANGEPRIORITY call.

8. Synchronization between threads is accomplished with events which function in much the same way as Freiburghouse's locks except that LOCK and UNLOCK are replaced by WAIT and STIMULATE and WAIT has nothing which corresponds to the ELSE group.

The RTL approach has the advantage that threads do not depend on each other for their existence. The only deficiencies of the RTL approach are:

i. There is still no time or time interval specification.

ii. Status checking is limited.

iii. There is no direct way to specify the procedure to be executed by a thread.

iv. RTL events are not quite as powerful as the Freiburghouse locks.

PEARL (Ref. 7)

1. A task is generated by an ACTIVATE statement.

   a. The ACTIVATE statement may have a WITH PRIORITY option to specify its priority. If this option is omitted, the task being created is assigned the same priority as the task which created it.

   b. Any task operation can specify a time with an AT option.

   c. Any task operation can specify a time interval with EVERY and UNTIL options.

   d. The code executed by a task is a statement or block associated with the ACTIVATE statement rather than an external procedure.

2. There is no direct way of delaying a task for a specified time period.

3. A task is suspended with a DELAY statement.

4. A task is unsuspended with a CONTINUE statement.

5. A task is terminated if: it and all its subtasks have executed their last statements; the task is referenced by an executed TERMINATE statement; or one of its ancestor tasks is terminated by a TERMINATE statement.

6. The only explicit status checking is with the PRIORITY built-in function.

7. The priority of a task can be reassigned by a CONTINUE statement with a WITH PRIORITY option.

8. There are semaphore variables which are accessed by REQUEST and RELEASE operations. These are almost equivalent to the locks, LOCK statement, and UNLOCK statement of Freiburghouse except that PEARL adds priority information. Also the results of a series of RELEASE's followed by a series of REQUEST's will be somewhat different than a series of UNLOCK's followed by a series of LOCK's.

The PEARL approach in general is very similar to the PL/I approach, with many of the same disadvantages. The differences are:

i. PEARL does allow specification of time and time interval.

ii. There is no direct way of delaying a task for a specified time period.

iii. The code executed by a task is not an external procedure but a block that appears in the activate statement.

iv. There are semaphores which are almost equivalent to locks.

v. Multiple activations of a task which would occur because an EVERY option has been used in activating a task can be prevented with a PREVENT statement.

vi. One task can wait for the termination of another task only by setting up a semaphore specifically for this purpose and RELEASEing it as the last statement in the second task.

LAI (Ref. 8)

1. A task is created by invoking a special kind of procedure called a "process." Each process has a waiting list of activations waiting for the task currently using the process to terminate.

    a. A priority can be assigned to a task when the task is created. If none is given, the priority declared statically by the process is used.

    b,c. No time or time interval is provided.

    d. The procedure executed by the task is the process invoked when the task was created.

2. A task can be delayed for a specified time and/or until an event occurs by executing a WAIT statement.

3. Any task can suspend any other task by executing a STOP statement naming the process being executed by the task being suspended.

4. A task is unsuspended by a GO statement which names the process the task is executing.

5. A task is terminated when it executes a RETURN statement in the process it is executing.

6. Tasks can check each other's status by means of events and semaphores. Events are special boolean variables managed by the supervisor. An event is set by an ACTIVATE statement and reset by an ERASE statement.

7. The priority of a task cannot be reassigned while it is executing.

8. LAI has semaphores which are exactly like those in PEARL except that the operations are called RESERVE and FREE instead of REQUEST and RELEASE.

The deficiencies of the tasking facilities of LAI are:

   i. There is no re-entrancy of processes, i.e., two tasks cannot be executing the same process at the same time. "Subprogram" procedures, however, can have the REENTRANT attribute.

ii. One task cannot terminate another.

iii. Priority cannot be reassigned during task execution.

iv. No time or time interval is provided for task activation.

However, LAI has several good unique features:

i. The WAIT statement can combine events and time periods.

ii. It has both events and semaphores.

iii. A task can receive a statically declared priority.

Hungarian Paper (Ref. 9)

This paper describes a process control language which is based on ALGOL 60.

1. A task can be created by calling a special kind of procedure called an "independent program."

    a. A priority can be established by assigning a value to a variable called CALLPRI.

    b,c. No time or time interval can be specified.

    d. An independent program is referenced by a program number rather than a name.

2. A task can suspend itself for a given length of time.

3. A task can suspend itself.

4. A task can be unsuspended only by an external event.

5. A task is terminated by executing an EXIT statement.

6. Tasks can check each other's status through variables called PRIORITY and STATE.

7. Priority can be reassigned.

8. There are no facilities built into the language for semaphores or task synchronization.

CONCLUSIONS

It is clear from the above summary that each of the functional requirements has been included in at least one language, but each proposed language is deficient in several of the functional requirements. Also, the same functions have been defined with entirely different terminology in different languages. Certainly, all the necessary work has been done somewhere by someone; all that is required is to merge all these proposals into one consistent language which meets all the functional requirements. The following proposal is an attempt to accomplish this.

PROPOSAL

1. Start with PL/I as modified by Freiburghouse.
2. Add the PRIORITY option from PL/I.
3. Add the ATTACH option from LTPL.
4. Add the AT, EVERY, and UNTIL options from PEARL.
5. Add a time specification to the WAIT statement as in LAI.
6. Since WAIT can now specify a time period, delete the DELAY statement.
7. Eliminate the STOP statement.
8. Add the PRIORITY pseudo-variable and built-in function from PL/I.
9. Allow a value to be assigned to the PRIORITY pseudo-variable while the associated task is active.

REFERENCES

1. Pike, H., "Tasking, Interrupt Handling, and the Synchronization of Parallel Computation Processes," LTPLC Draft Position Paper, April 6, 1971.

2. PL/I Language Specifications, IBM Form Y33-6003-1.

3. Beech, D., "A Structural View of PL/I," Computing Surveys, March, 1970.

4. Freiburghouse, R., "A Proposed Tasking Facility for PL/I," ANSI Working Paper X3J12/80, February 26, 1971.

5. "Report of the Long Term Procedural Language Committee," Minutes of Fourth Workshop on Standardization of Industrial Computer Languages, November, 1970, pp. 103-139.

6. RTL Project Report No. 1, RTL Definition, Imperial Chemical Industries Ltd., August 1, 1970.

7. "PEARL, The Concept of a Process-and Experiment-Oriented Programming Language," Fourth Workshop Minutes, pp. 162-175.

8. "Industrial Programming Language: LAI," Fourth Workshop Minutes, pp. 145-153.

9. Gertler, J., "High-Level Programming for Process Control," Third Workshop Minutes, pp. 147-152.

- L T P L C -

WORKING PAPER

T A S K I N G

P. RIEDER

OCTOBER, 1971

INTRODUCTION

This paper is intended mainly to revise and complement the working paper of P. Lindes (ref. 2) with particular regard to the PEARL approach. The papers by J.G.P. Barnes, R.J. Long, D.N. Shorter (ref. 3) and H. Pike (ref. 1) are also taken into consideration, but not to the same extent. Special emphasis is laid upon some problems where PEARL seems to offer preferable solutions.

A CHECK OF PEARL AGAINST THE PURDUE FUNCTIONAL REQUIREMENTS

The recently revised (not yet published) version of PEARL fulfills all functional requirements listed in reference 2. An examination shows (numbering according to 2):

1) All statements containing no jump outside can be established as tasks. This implies as a special case the call of global procedures. The code to be executed in a task may be specified either within the ACTIVATE statement or alternatively with the declaration of the task

name. A task is attached to the innermost block containing the statement where its code is specified, this block possibly being the system itself.

a) By an option PRIORITY a priority may be specified. This priority is understood to be relative to the task to which the newly created one is attached or alternatively relative to the system. To distinguish these two possibilities the keywords REL and SYS are used. Priorities relative to an ancestor task (sub-priorities) render possible a tree-like priority structure.

b),c) Any task operation can specify time(s) and/or time interval(s) by AFTER, AT, ALL, EVERY, DURING, UNTIL options. Thus schedules like

AFTER 1 HOUR ALL 15 MIN DURING 6 HOURS

or

AT 9:30:00 EVERY 2 MIN 30 SEC UNTIL 22:12:30

may be arranged.

d) If the code of a task consists of a procedure call then it of course contains a procedure name. Each task must have an explicitly declared task name.

2) A task may be delayed with automatic continuation by the statements e.g.

SUSPEND TASK1 UNTIL 13:00:00;

or

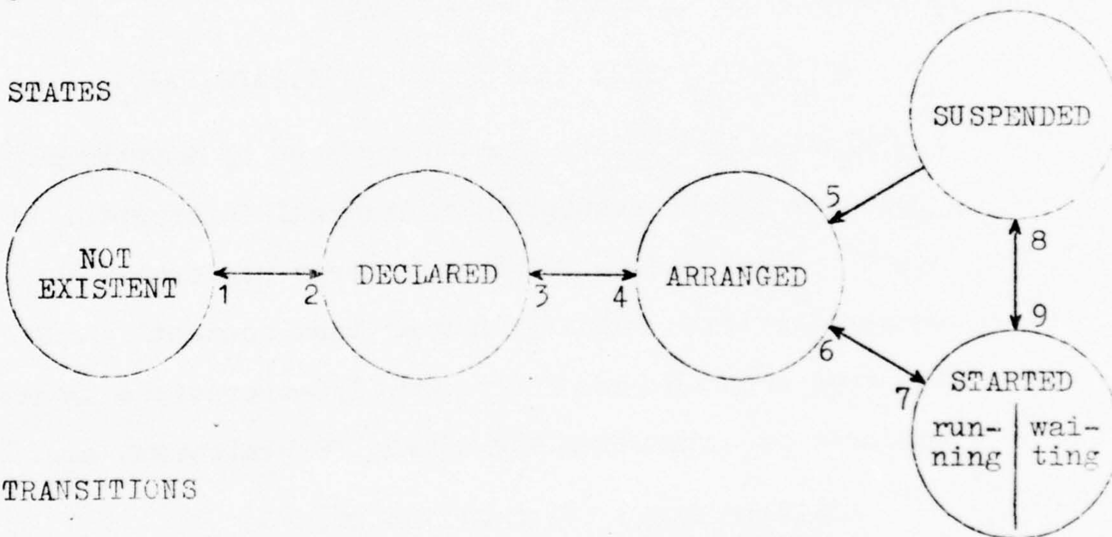
SUSPEND TASK1 DURING 7 MIN;

3),4) A task may be delayed and unsuspended on demand by the statements e.g.

SUSPEND TASK2;   CONTINUE TASK2;

5) A task ends, if it and all tasks attached to it (subtasks) have executed their last statement. It can be forced to an irregular termination by an explicit TERMINATE statement thereby also simulataneously ending all its subtasks. If a task is arranged by a schedule to be initiated only later on, this initiation can be presented not by TERMINATE, but by a special PREVENT statement.

6) PEARL defines several states of a task and implicitly or explicitly demanded transitions between them, as the diagram shows:



TRANSITIONS

1 obliteration	4 disposition	7 initiation
2 declaration	5 termination	8 suspension
3 cancellation	6 termination	9 continuation

No direct means to check these states are deemed necessary. The priority of a task, however, may be tested by a special built-in function.

7) The priority of a task may be changed during its runtime by the statement e.g.

CONTINUE TASK1 PRIORITY K;

8) Semaphore-variables supply more synchronization possibilities than lock-variables do. Probably the most important of their advantages is, that the contents of a semaphore depends only on the number of executed REQUEST and RELEASE operations, but not on their chronological sequence, which in real-time applications is rather unpredictable.

DISCUSSION OF SOME PARTICULAR PROBLEMS

1) The concept of tasking is too basic in a language for process control purposes to be implied in optional appendices to the CALL statement: LTPL should provide a separate statement for creating tasks, e.g.

```
ACTIVATE TASK1: CALL PROC1(PAR1,PAR2,PAR3);
```

2) Parallel activities are set up also by asynchronous I/O-processes and by responses on external interrupts. With respect to language uniformity it seems preferable also to include these activities into the normal task concept. In PEARL the asynchronous performance of I/O-operations is readily achieved by activating the usual I/O-statement, e.g.

```
ACTIVATE TASK1: READ VARIABLE1;
```

To activate interrupt responses a special schedule element ON is provided, e.g.

```
ON INTERRUPT1 ACTIVATE TASK1: CALL INTERRUPTRESPONSE1;
```

Different mechanisms to suspend the execution of a routine eventually supplied by the OS are employed by the implementation, but not reflected in the language.

3) If as in PEARL tasks are declared explicitly the maximum number of simultaneous tasks can be checked at compile-time

and used by the compiler to provide appropriate space for housekeeping information. Declaring a task does not necessarily imply loading its defining code, even if specified within the declaration.

4) In the LTPL proposal (like in PL/1) the normal ending of a task causes the abrupt termination of all its subtasks. This is very dangerous in process applications where depending on the varying charge of the disposable processors sometimes the subtasks and in other circumstances the attaching task may come earlier to their natural end. Thus the real-time performance of the system remains undefined. To avoid this PEARL chooses the reverse possibility: The end of a block implies waiting for all tasks attached to it.

5) Regarding the above consequences of attaching it seems necessary to distinguish the block to which a task is attached from the block which contains its activation. In LTPL this is achieved by an ATTACH option in the CALL statement naming the task the newly created one shall be attached to. As a special case thereof a task may be attached to the system. Using the ATTACH option implies, that not the actual parameters themselves but copies of them are used in the attached task. Apart from the fact, that copying of e.g. large arrays may cause a considerable loss of storage space and execution time, the different passing of the same arguments to the same procedure depending only on the ATTACH option is rather misleading. A great unsolved problem in this approach is how to handle variables and procedures declared in a procedure surrounding the called one. Copying all these entities too seems rather inefficient and unreasonable. The solution provided by PEARL is more natural and

expedient: A task is attached to that block where its code, e.g. a procedure call, is specified; thus no contradictions between the scope of the entities used in it and their needed lifetime can arise. On the other hand, the specification of the code to be executed in a task may be separated from its activation; in this way a task attached to the system, for instance, nevertheless may be activated within any other task.

6) Generalizing Dijkstra's semaphore operations, REQUEST and RELEASE may be applied also to lists of semaphores. Beyond the effect of separately performed REQUEST operations on several semaphores, in PEARL

REQUEST S1, S2, ... S_n ;

results in checking simultaneously all semaphores contained in the list. The operation can be completed only if all affected semaphores at the same time have non-negative contents.

RELEASE S1, S2, ... S_n ;

means releasing all listed semaphores simultaneously. Semaphore operations on lists supply a very flexible means for synchronizing co-operating processes.

7) The exchange of messages may be achieved by the shared use of global variables. Access by incompetent other tasks is easily prevented by the use of unique names. PEARL facilitates this exclusion by so-called link-names, link-time supplements to the identifiers of global entities available within a particular program. In separate programs equal names are considered to represent the same thing only if also

the same link-name is applied.

8) Semaphores and shared global data seem widely accepted as an efficient means for synchronization and message exchange. In RTS/2 a considerable variety of additional higher-level communication and synchronization functions is provided. Before they all are included in a LTPL standard it seems necessary to investigate how much security and comfort is gained and how much efficiency and flexibility is lost by their use.

REFERENCES

1) Pike, H. "Tasking, interrupt handling, and the synchronization of parallel computation processes". LTPLC position paper, April 1971.

2) Lindes, P. "Task interactions". LTPLC working paper, April 29, 1971.

3) Barnes, J.G.P., Long,R.J., Shorter, D.N. "Tasking". LTPLC draft position paper, September 1971.

- LTPLC -

POSITION PAPER

TASKING

J G P BARNES, R J LONG, D N SHORTER

OCTOBER 1971

1 INTRODUCTION

This paper is to a large extent a critique and extension of the working paper prepared by P Lindes (Reference 1). It includes, as Appendix 1, a paper by J R Prior of the Royal Radar Establishment, which was presented at a recent symposium in Harwell, England, and which seems to us to provide an excellent survey of the requirements for task synchronisation. The second appendix contains a description of task states and a tabular comparison of various languages; these were developed at a recent meeting of the European group.

2 CRITIQUE OF LINDES' FUNCTIONAL REQUIREMENTS

2.1 Scope

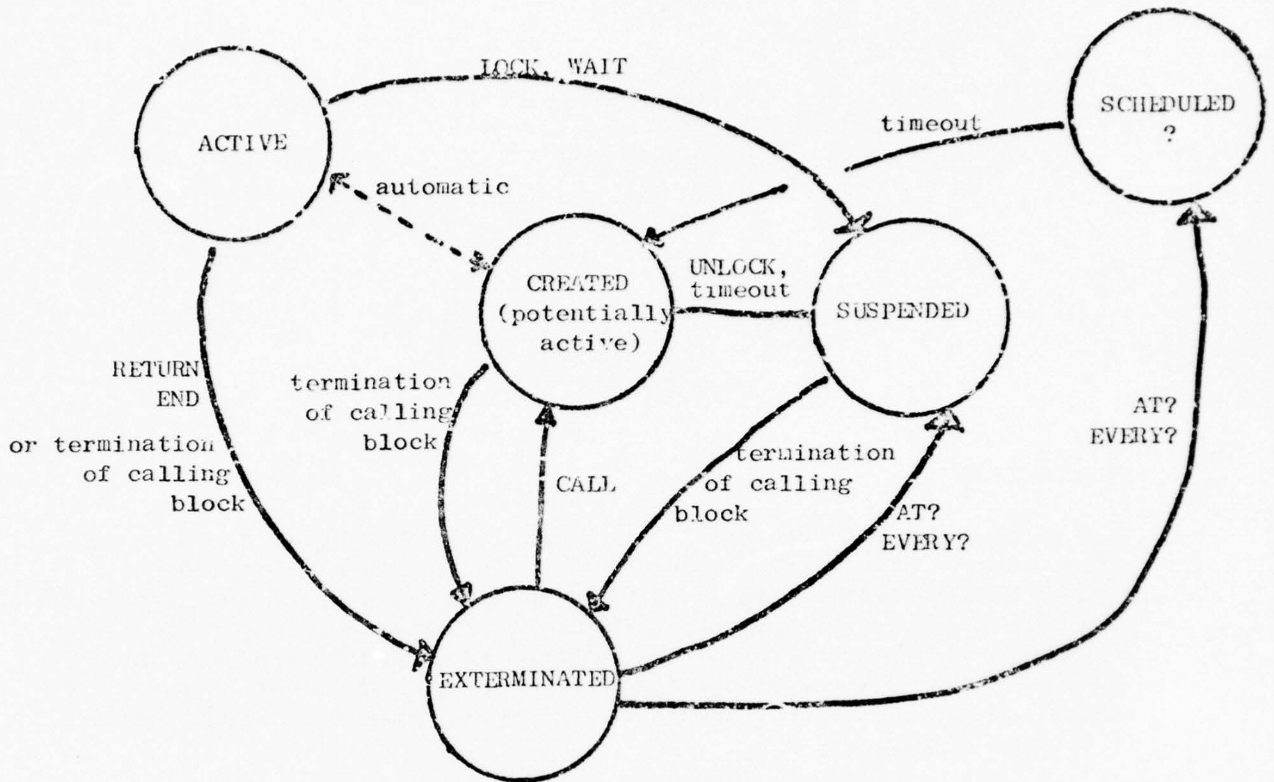
This section covers two areas discussed in some detail by the European LTPL group.

2.2 Task State Changing

The European Group has spent some time analysing task states and state changes in different languages or systems - the results of some of this work are shown in Appendix II. The Group consider that it is essential to be able to draw out a clear state diagram illustrating states of interest to a user, and the system or language calls available to him to change these states. It must be clear what features of changes are defined in the language and which are implementation dependent (e.g. are they immediate or do they await, for instance, a core to backing store program swap). Such a task state diagram also enables the syntactic form of the state changing statements to be cast in such a way as to place the right weight on the most frequently used options.

The Group found that they had insufficient information to draw out a state diagram for the Lindes proposals or for the revised PL/1 proposals. A first attempt to cast Lindes' proposals into a state diagram form resulted in the following:

PRECEDING PAGE BLANK-NOT FILMED



Lindes Proposals State Diagram

The states shown here do not directly correspond to those of Appendix II. A particular difficulty was encountered in deciding the status of 'AT' and 'EVERY' which may generate a new 'SCHEDULED' state.

2.3 Synchronisation

The primitive synchronisation facilities suggested are not suitable for use in all applications. The paper by Prior (attached as Appendix I) analyses interactions in more detail and techniques for handling them. We would recommend serious consideration of his conclusions.

The synchronisation system must be inherently robust, by allowing the program complex to recover from system malfunctions. In particular, a "timeout" facility is often required for hardware events (e.g. reading a card). Timeout is also required for message and event handling, and the facilities for hardware and software synchronisation should be identical in concept. One might therefore propose that semaphores, events and messages should all have a timeout facility returning an error flag or jumping to an error label when this occurs, as a possible primitive solution for these problems.

The current PL/1 proposals for LOCK/UNLOCK have an ELSE facility which allows a timeout facility to be implemented, albeit clumsily.

3 MODIFICATIONS TO LINDES' LANGUAGE REVIEWS

3.1 Scope

The following sections are included by way of extensions and corrections to Lindes' paper. They are organised to relate to his original functional requirements. A more detailed analysis of state changing in these languages is given in Appendix II.

3.2 RTS/2 (Refers to the section on RTL, Lindes, page 6)

In Lindes paper, the document used for the review of RTL presented a rather primitive interface, which was not the one seen by the users. For example, to the user, makethread did specify the procedure name, and secure and release were also available.

RTS/2 is a system designed for the language RTL/2, and differs from the earlier version (RTL/1) in Lindes' paper. RTL/2 is, among other things, a language for writing operating systems, and RTS/2 is programmed in the high level language itself. The functions described here are supervisor calls in RTS/2 but they are not fundamental and can readily be modified - they are only a recommended set for medium scale applications. The language can run under a wide variety of operating systems.

In RTS/2 the conventional term "task" replaces the"thread" of RTL/1.

3.2.1 A task is created by a MAKETASK call but does not start executing until started by a START call.

a Priority specified as a parameter of MAKETASK
bc No time or interval can be specified
d A procedure name is specified as a parameter of MAKETASK, together with a dynamic work area ("stack") to facilitate re-entrancy in all procedures

A further parameter specifies buffer requirements for message transmission by the task.

3.2.2 The same DELAY as for PL/1, but with time units implementation-defined.

3.2.3 A task may STOP itself or any other task indefinitely. It may become suspended:

(i) While awaiting the release of a Dijkstra-type semaphore ("SECURE")
(ii) While awaiting the arrival of any signal ("IDLE") or any message ("LISTEN") from another task.
(iii) While awaiting the arrival of a signal ("WAIT") or a message ("ATTEND")from another specified task.

In cases (i) and (iii) a time out exit is available, as an aid to system security.

3.2.4 A task may become reactivated by being STARTed by another task (processing continues from point of stoppage). It leaves the appropriate suspended states when a message is sent to it ("SEND" by another task) or a signal is sent to it ("STIM"), or if a semaphore is RELEASED, or on time out of SECURE, WAIT or ATTEND.

3.2.5 A task may exterminate itself or any other named task by a "KILL" call. It is also exterminated if and when control leaves the main procedure in the normal way.

3.2.6 Tasks can find their own priorities by an OWNPRIORITY call. Other status information is held in shared data areas or communicated in messages.

3.2.7 The priority of any task can be reassigned with a CHANGEPRIORITY call.

3.2.8 Synchronisation between tasks is accomplished normally by message transmission or by signal transmission (a signal being a message without any content other than the sender's identity). Where efficiency demands it, Dijkstra semaphores are used. It is considered that the use where possible of messages rather than shared facilities protected by semaphores provides a system in which user mistakes are less likely.

The RTS/1 approach, like RTL before, allows tasks to be independent of each other for their existence. No direct facilities for task scheduling are available, although such could readily be added.

RTS/2 differs from other systems in Lindes' paper in that a considerable variety of task communication and synchronisation functions are included; these are considered to be one of the most vital aspects of real time programming, particularly from the point of view of long term security.

Events have been deleted from RTS/2.

3.3 PEARL (Lindes, page 7)

Since Lindes paper was written, work on an improved version of PEARL has been accomplished and is just being written up. The language now fulfils all functional requirements mentioned in H Pike's paper (reference 3). The main improvements to the previous version are:

- Newly created tasks may be attached to the system. This has the consequence that the life time of a task may be longer than the life time of the task which created it.
- There is now a direct way of delaying a task for a specified period of time (e.g. <u>delay</u> taskname <u>during</u> 5 <u>min</u>),

3.4 SINTRAN

This is a language not covered by Lindes' original review. For a description see reference 2.

3.4.1 A task is created and activated by executing a CALL SET, CALL ABSET or CALL RT statement with parameters given below. The existence of a task does not depend on that of the creating task.

a The priority of a task is that of the program (see d). A task is started only if its priority is higher than that of any other using the stack (used for workspace). This priority may be changed by a CALL INHIB for a temporary change (followed by a CALL FREE to reset) or by a CALL PRIOR for a permanent change.

b A task can be started at absolute time hour.min.sec. by a CALL ABSET (prog,sec,min,hour) statement.

c A task can be started after a delay, by a CALL SET (prog, time,units) statement.

d When a task is created it executes the program named in the corresponding creation statement (e.g. CALL RT(prog)).

A task can be scheduled for periodic creation and activation by CALL INTV (name, interval, units). The effect is to schedule the activation of program 'name' at periodic intervals. The first activation must be by a CALL RT (name) statement.

This periodic execution will continue until a CALL DSCNT (name) statement is encountered.

A task may be created (and activated) as a result of a hardware interrupt, by executing a CALL CONCT (name, interrupt). This convection will continue until another CONCT call is made, or until DSCNT is called.

3.4.2 A delay may be implemented in a program P by a CALL INTV (P, delay units) followed by a STOP. On re-entry the program P must CALL DSCNT(P). During this delay the stack space (i.e. the working storage) will be lost.

3.4.3 A task may be suspended by a PAUSE statement, awaiting operator action (with no loss of stack space). No event statements are provided.

3.4.4 A suspended task (program XX) may be reactivated by the operator command CONT XX

3.4.5 A task (program P) may be exterminated by encountering the END of a Fortran master segment, by encountering a STOP (optionally STOP YY) statement, or by the execution of a CALL ABORT (P). If P is not active, then ABORT has no effect.

3.4.6 No direct facilities are available for checking on the priorities or status of programs.

3.4.7 Reassign priority - see 3.4.1a.

3.4.8 Data access interaction between programs is prevented by temporarily raising the priority by INHIB.

This system has the disadvantage of not having DELAY/CONTINUE statements, synchronisation or event variables.

4 RECOMMENDATIONS

The Lindes' proposals taken with those of Prior and suggestions made above, would form a good starting point for a tasking investigation team. While appreciating that a considerable amount of work is involved, it appears essential that these proposals be formulated in detail, in a single document, so that redundancies and inconsistancies can be identified and to ensure a common style. This document should include a task state diagram, or equivalent.

5 REFERENCES

1 Lindes P. "Task Interactions". LTPLC Working Paper, April 29 1971

2 "Sintran II User's Guide", A/S Norsk-Data-Electronikk, June 1 1971

3 Pike, H. "Tasking, interrupt handling and synchronisation of parallel computation processes". LTPLC Draft position paper, April 6 1971.

APPENDIX I

PROGRAM INTERACTIONS IN REAL-TIME SYSTEMS

J R PRIOR

1 INTRODUCTION

It is becoming increasingly apparent that one of the major impediments to the implementation of large real-time programming systems is the difficulty of obtaining high software reliability. There are of course a variety of reasons why such systems can fail to meet their original aim. These can vary from insufficient analysis of the problem to simple programming errors. The aim of this paper is to discuss one of the types of programming errors and to review the various techniques used to avoid these errors.

Figure 1 shows a diagram of a typical real-time computer based system. It shows how such systems are composed of data sources and sinks, man-machine communication facilities and a process to perform the necessary data transforming actions. In any practical system the process will be sub-divided into many small processes. This subdivision allows many programmers to work on the program development since each can work on separate program modules. It also allows more flexibility in the design so that the eventual program package can meet the demands of response time, variable priority of function, and modifiability. Since the complete functions of the system are then subdivided, and implemented by a number of program modules, it is necessary for these modules to co-operate together. This co-operation has two aspects, the control of activation of the processes to perform the overall job, and the sharing and communicating of data between the processes. System errors arising through failures in this co-operation are of particular concern as far as the reliability of real-time systems are concerned since they are often very difficult to locate. They tend to occur only infrequently when an unlikely order of running occurs between processes, furthermore the erroneous data output may be detected only after the originally faulty processes have resumed correct operation, thus making it difficult to trace back to the original fault.

DATA TYPES

Data sources very widely in their characteristics but they can be subdivided into two classifications, (1) those which provide messages and (2) those which provide status information.

Figure 2 shows in more detail the type of program structure one might expect for handling a message data source. Here the external data source repetitively produces messages, each perhaps consisting of several words of data. These messages are handled by processes which transform the data according to preset rules, defined by the algorithms, but varied by the data read from other processes.

The usual characteristics of message exchanges between processes are as follows:

a) Each message is used at least one process; hence activation of the process may be arranged to take place as a direct result of the generation of the message.

b) Messages are not of immediate importance hence they can be buffered.

c) Message data is transient. It is "consumed" by the processes using it i.e. These processes destroy the message.

The second type of data source produces status data. This data conveys information about the state of the surroundings of any process which may be of use to the process as it works on messages. Status data may be generated by external devices or by processes.

The characteristics of status data exchanges are as follows:

a) Status data is non-transient. It is not "consumed" but is changed by the processes generating it. It is frequently "updated" to a value dependent upon its old value.

b) Status data is of immediate importance since it supersedes the previous status data, hence it must replace this.

c) Status data may be used many times or not at all. It does not cause processes to be run directly.

In real-time systems messages arrive from the external hardware devices and flow through various processes into a pool of status data. This pool contains status data about many independent entities, such as aircraft for example. Messages are extracted from this pool and sent to output devices. Whereas the individual messages are treated independently, the actions taking place on the pool of status data often involve complicated comparisons between status blocks relating to different entities. Many difficulties in real-time programming occur at the interfaces between messages and status data pools.

INTERACTION TYPES

At any data source-to-process interface in the system, whether it be hardware or software, the same type of operations take place. The data transfer from the external device into the computer store is performed by the use of control wires which ensure correct co-operation between the device and the computer highway. In a similar fashion exchange of information between a software data source, a data table, and a process involves not only the passing of data but also the use of controls.

A hardware data source may be polled or it may demand attention by the use of a hardware interrupt line. The software data exchange between processes P_1 and P_2 of Figure 2 illustrates a parallel situation since the buffer may be polled by P_2 to find if data has arrived, or P_2 may be activated via the scheduler on a demand from process P_1. The activation lines are not shown in Figure 2.

The fact that both data and controls are involved in communication between processes allows two types of interaction error. Firstly, those which arise through incorrect use of the controls giving access to the data. Then one might for example access beyond the bounds of an array or read data from an empty buffer. Such interactions will be called macrointeractions. Secondly we have situations in which any data block is read whilst not fully specified. Thus only part of a correct message is read. These interactions will be called microinteractions.

2 CONTROL OF INTERPROCESS COMMUNICATION

The aim of many methods used to control destructive interference in real-time programs arising via message or status exchanges has been to attempt to restrict all interactions to macrointeractions.

In the sections below we will consider some of the methods that have been used in current systems to prevent interaction errors. Any method of control can be judged against the following check list.

1) Is the system response time adversely affected? This will be so if the control method is expensive in program length or if it prevents the processor working on the most important tasks.

2) Can the technique be implemented reliably? Preferably control of interprocess communication should be in the hands of the system designers and not the coders.

3) Is subsequent program modification easy? This will be so if the method is formalised and well documented.

4) It should be reasonably efficient in the use of core and time.

5) It should control both micro and macro-interactions on both status and message data exchanges.

6) Can operating system checks be employed? If they can be used then program development is more easy, and subsequent extensions and modifications can be performed more rapidly.

7) Can storage protection techniques be used effectively? The extent to which protection can be used depends on the hardware facilities to a large extent but it also can depend strongly on the communication method used.

2.1 Running to completion

The most obvious way of preventing microinterference between processes is to run all processes to completion i.e, once a process had started it is uninterrupted by any other process. Therefore, at no stage is there any chance of a process leaving a data area incomplete. If one is to maintain a flexible system design capable of giving a rapid response to high priority inputs, one must ensure that the complete program is subdivided into many small processes each of which is called independently by a scheduler . Hence the maximum delay in responding to any new external event is that of running the longest of the processes. If each process leaves the data areas it acts upon in a complete state then the possibility of microinterference is limited to situations like that indicated in Fig. 3B. In this example situations can occur where the data input areas of P_3 are incomplete unless both P_1 and P_2 have completed their cycle of operations. One therefore must ensure that the running order of the process is $P_1P_2P_3$ or $P_2P_1P_3$. Interaction E is another example where microinteractions can occur.

Besides largely removing the problem of microinterference, running to completion does allow subroutines which are common to many processes to be shared freely without re-entrance problems.

The macrointeraction problems can be solved in a number of ways. For example consider interaction A in the diagram. This shows a typical interaction involving the exchange of status or message data between processes. If for the moment we assume that this is a status exchange and that both processes run to completion then macrointerference can be avoided by ensuring that P_1 operates before P_2 so that as the system starts operation there will be status data available for P_2. This may be achieved by associating a correctly initialised control flag with the data area and ensuring that P_1 and P_2 observe this flag. Alternatively the jobs may be called by the scheduler in the correct order, initially. The scope for macrointerference is thus rather small. If the exchange should be of a message nature, the data area must become a buffer to allow for uneven rates of production and consumption of messages by P_1 and P_2. This macrointerference can be controlled by defining a set of input and output pointer manipulation rules and allowing P_2 to poll the buffer to determine whether there are any unprocessed messages. Such pointer rules must allow for the situation when the list is full or empty. For the more common data structures the individual programmers can be protected from the problems of reading and writing into the data areas by the use of standard routines. These can be part of the applications programs but many systems, as will be described later, use the operating system for this purpose. For more complicated data structures in which many processes add or subtract data, it is preferable to have a common subroutine to handle the input and output of data to the data area, rather than allow each process to do so independently.

If the technique of running to completion is used throughout a system, it implies that the external devices must also be polled. Since the variable data input/output rates and response times of external devices frequently necessitate the use of an interrupt response to these devices, many systems employ a minor variant of this technique by restricting actions stimulated by an external interrupt to the input or output of data. If the data is of a transient (message) nature it can be stored in lists. Microinteractions then only take place at these lists as the interrupt processes feed data in and the base level processes poll the lists to determine whether any data has arrived. Careful selection of the rules governing manipulation of the pointers giving access to the buffer can remove both micro and macro interactions on this buffer area, by ensuring that updating of the control pointers to the list takes place on a single uninterruptable action. Besides governing access to the lists these rules must allow for control of the input data rate when the list becomes full, if this is possible. Alternatively, if the data source cannot be controlled, and all the data is required, the input buffer length must be dynamically varied or the consumer process must run with high priority. A further point to be considered in organising these lists is the method used to start up and stop the external device, since depending on the nature of the device, this should either be done in the interrupt program or in the base level program.

Should the data arriving "on interrupt" be of a status nature and consist of several words it is necessary to prohibit interrupts whilst the status block is being changed since all the new status words must replace the original words before any process is allowed to read the status.

As an alternative to the base level programs polling the input buffers. it is possible to link the arrival or messages from external devices with the activation of the consumer process. Thus the interrupt program can demand the activation of the consumer process. This can lead to a reduction in the input buffer sizes whilst still allowing higher priority base level processes to have priority.

2.2 Multiprogramming systems

In spite of the advantage of running to completion as an aid to simplification of interaction problems, there are real-time systems in which this approach cannot be used. Some of the reasons why this is so are summarised below.

1) High response time requirements may demand that a process be interrupted before it has completed its change to data areas used by other processes. For this reason processes should be kept short. This is not possible where such processes require input data and the input buffer is empty or require to output and the buffer is full. Then the process must await the response of external devices thus increasing response time and wasting processor time.

2) If the system is not dedicated to one job, multiprogramming is required to ensure that processor power is distributed so that good response times and throughput on each job are achieved.

3) If co-equal multiprocessing is used, where any available processor can run any program, the methods of interaction control used in running to completion are ineffective.

For the above reasons a multiprogramming mode of operation is required in which the individual base level processes can interrupt each other as the scheduler reallocates the processor from one task to another. The scope for microinteractions then spreads from those processes dealing with hardware interrupts throughout all the interacting processes. Furthermore the interaction problem becomes more complex than at the hardware interface since now each process acting on a data area can interrupt the other.

2.2.1 Lockout Techniques

In multiprogrammed systems of this type many techniques have been used to control interactions. During access to a shared data area any process may prohibit external interrupts during the critical section of its program in which it is changing the data area. Such a technique is unnecessarily restrictive in that other unrelated processes are also prevented from running and data arriving from external equipment is delayed or even lost.

An alternative somewhat more efficient method is to place the critical section of the task which is currently modifying a data interface area on the highest scheduler priority until the data area is complete. Thus external devices are not held up, though once again other base level processes are impeded. Neither of these techniques are suitable for use in the multiprocessor case since they rely on ensuring that only one process is operating during the critical part of the process, nor do they provide a solution to the interaction of figure 3B unless the critical section extends across more than one process.

Another secure method which can be used in multiprocessors is that in which a table of interacting processes is kept within the operating system. The scheduler then ensures that at no time do two interacting processes become active at the same time. This method is easy to implement and allows modification of programs without unforeseen errors arising. New processes can also be added since the table of interactions can be easily modified. The method is however more restrictive than is strictly necessary since processes are prevented from operating from longer than they need be.

Semaphores are perhaps the best known method of controlling microinteractions. These are a very general method of protecting critical sections of a process from interference. A control flag is associated with any shared data area device or process. Such areas, devices and processes are called facilities. All processes using a facility examine the flag to ensure it is available for use. Although they are simple in use and are efficient, in that lockouts are restricted to those processes requiring simultaneous access to the same facility, the individual programmer is responsible for their proper use. The binary semaphore (in Dijkstra's terminology) has been most widely used to prevent read-write interactions between single processes. Unnecessary lockouts can be reduced by permitting parallel read access to a data area whilst excluding read-write accesses. Semaphores have the advantage of imposing little overhead on most processes since they are used only where necessary and can be implemented quite efficiently on most modern computers. They can be used effectively in systems using co-equal multiprocessing.

The semaphore mechanism can also be used as a means of activating processes when data has arrived for them. This is done by a technique described fully by Dijkstra[1] in which the producer process "releases" the data area when it has provided the data and the consumer "secures" the area before it begins to read. In the absence of data the consumer will then be suspended until some data arrives.

Although the mechanism controls microinteractions on single data areas, where access of the type shown in Figure 3C is concerned there are difficulties. If both data areas must be changed by process 1 before process 2 can operate, it is necessary for process 2 to secure the two facilities simultaneously. It is not enough to declare a single facility equal to both lists in addition to those controlling the individual lists since the access to the lists would then be controlled by two independent controls. Although the situation of Figure B can be controlled by the use of three semaphores, one on D_1, one on D_2 and one on D_1 and D_2 this is cumbersome in use and to deal with the general situation the ability is required to "secure" many facilities in a single uninterruptible action. In addition to this requirement for simultaneous securing of many data areas, it is also advantageous to provide a selective release so that lockouts of other processes can be minimised.

One of the problems to the use of semaphores is the risk of "deadlocks". One such example of these is the "deadly embrace". This occurs in situations like that of Figure 3C when P_1 uses the data areas in the order D_1, D_2 and P_2 uses them in the reverse order. It is then possible for each process to hold a data area required by the other. This problem can be resolved by numbering all facilities and then ensuring, via the operating system, that facilities are always demanded in numerical order when more than one is required simultaneously. Though solving the problem this does mean that in some cases facilities will be held for longer than is really necessary.

An alternative form of deadlock occurs in situations like that of diagram A where P_2 is a procedure used by P_1. If P_1 acquires D before transferring control to P_2, P_2 is unable to run hence P_1 cannot complete.

Habermann[2] describes a technique for avoiding deadlock in which advance warning is given to the operating system whenever a process is going to require more than one facility simultaneously.

A limitation of the semaphore mechanism is that it cannot be used to control interactions between the base-level and interrupt programs since the attempt to secure a data area involves the risk of suspension. This generally cannot be allowed to happen to an interrupt program.

2.2.2 Data Base I/O routines

The foregoing methods have all, with varying degrees of efficiency, controlled the microinteractions but left macrointeractions to be controlled by agreements reached between the programmers working on the project. There is a serious danger that those who later maintain the systems will not be familiar with the conventions used and may consequently generate perplexing bugs as a result of apparently trivial modifications. This danger is so acute that in systems of any complexity it may be best to allow access to data only by "system" routines.

One possible method for doing this whilst ensuring that the common data is not accessed freely by all processes is to eliminate the common data base. In such a system read-only data becomes the only globally available data.

Access to each individual array, table, buffer of the data base is controlled by a data area input/output routine for which the data is local storage. Such input/output routines will transfer data between the local working space of a process and the various arrays, etc. Dependent upon a parameter handed to it, the I/O routine will provide any of a number of options to the user. For example items may be added to the head or the tail of the list, or items above a certain magnitude may be removed. One must also allow for items to read and deleted or read without deletion. Many other options can be provided as required. Since only one routine is accessing any area of the data the scope for macrointeraction is greatly reduced. Figure 4 illustrates the form of the program structure. Microinteraction must be eliminated by ensuring that the data I/O routines can only be used by one process at a time. In fact such routines should be called by macros which ensure; for example, that the following sequence of operations always occurs:-

```
SECURE          (list 1)

CALL, I/O-1     (next entry)

RELEASE         (list 1)
```

where the process requires to read or write a new item, as for example when dealing with transient data, the call may be of a simple type and the I/O routine can be simple. However, when dealing with non-transient data which is being modified, the call must be more complicated since one needs to read, modify and write new values. This can be done by writing a complicated I/O routine incorporating this ability, but it is better if the call has, as a parameter, the procedure which modifies the data.

For example one could define a macro of the form:-

```
SECURE          (LIST 3)

CALL, I/O-3     (next entry, PROCEDURE P3)

RELEASE         (LIST 3)
```

where P_3 performs a modifying action on the "next entry" data extracted from list 3 by the I/O routine.

A scheme of the above type has the advantage that it is easy to enforce on programmers and easy to understand. The scope for faulty interactions is minimised and the system overheads are small.

2.2.3 Message systems

The way in which message passing is implemented depends upon the type of hardware storage protection available, the extent to which individual programmers can be relied upon to generate only legal messages, and the efficiency requirements of the system.

Message systems are an extension of the previous technique. The characteristics are that interprocess messages are passed using a dynamically allocated pool of storage administered by the operating system. In some cases the operating system merely allocates the storage and transfers it from process to process whilst in other cases all I/O to the storage is controlled by the supervisor.

Message systems can offer the greatest security against interference since neither code nor data need be shared. Diagram 5A shows the program structure of a reliable system. Each process is self contained with its own data space and constants. All communication with other processes takes place by messages passed by the operating system. A message can be assembled by P_1 in its own area then transferred by the operating system to the operating system area. The operating system queues messages for any process. These queues can either be polled or the accepting processes can be activated when messages are waiting. The message is again transferred to the accepting program by the operating system. Storage protection can be arranged to protect each process and its data areas from other processes. The operating system can apply checks on the legality of message interchanges so that messages only transfer between allowed processes, and also ensure that messages in the common message store do not mutually interfere.

Clearly a message system of this type is inefficient since data is moved several times. A slightly less secure method which is possible where application programs have read-access to the operating system area is to allow the accepting process P_2 to have "read" access to D_{op} so that it can read the message directly from that part of store.

The most efficient, though least secure mechanism is that shown in Diagram 5B. Any process wishing to communicate with another process asks to be allocated a block of store administered by the operating system. This area can then be used to store a message for transfer to another process. When the message has been generated the producer process P_1 asks the operating system to transfer the message to the accepting process. Once this request is made the producer process loses access to the message store which becomes linked to a chain of similar areas to await processing by the acceptor process. Since users have direct access to the common message area, it is possible for them to overwrite other messages, because with simple storage protection it is not possible to protect this area from all users. More sophisticated hardware protection techniques, in which read or write access could be given to processes for small, variable sized, areas of core, would enable this type of message system to be made more reliable. Such a scheme is provided by the Capability mechanism (3).

In order to give flexibility, various forms of entry and exit to the chains of messages can be allowed. For example one can set up LIFO, FIFO and "content" ordered chains. If the chains of messages are organized so that the messages themselves are chained together as in diagram 6A one can have a system that is efficient in passing messages in the situation shown in Figure 3A. However, in the situation shown in Figure 3D, one would need to pass three identical messages to the process queues for P_2 P_3 and P_4 with consequent core and time penalties. It is therefore advantageous to organize for each process a list of pointers to the messages awaiting it as in diagram 6B. In this way the situation of Figure 3D can be solved by merely adding the same address pointer to three lists whilst the data is only stored once. Where such a situation occurs the operating system must ensure that the message cannot be destroyed until all consumer processes have operated.

If efficient use of store is to be achieved it must be possible to pass messages of any length between processes since many processes generate messages of variable length.

What these systems have done of course is to hand over to the operating system all the problems of pointer manipulation and control of microinterference.

Message systems such as these in which a common store area is shared for communication between many processes are clearly more efficient in core utilization than the use of fixed length buffers. However, should this area ever become filled, all those processes generating messages will be inoperable. Hence the amount of free area should be monitored so that message "consuming" processes should be run preferentially whenever the free area becomes small.

Message systems are a suitable way of passing transient data about the system. However, it is not ideally suitable for status data. Such data is of immediate significance hence it must not be queued. Out of date status information must be destroyed by the producer process, rather than by the consumer, as is the case for messages. Whereas messages most often are sent to a single process, status data is frequently read by many processes thus requiring high overheads. Updating operations on status data cannot be handled efficiently by the message system.

For this reason the message system cannot be used exclusively. One way of mixing message systems with other forms of inter-process communication is to subdivide the complete system into groups of processes. Within any group the processes can run to completion and interact via shared data areas. The individual groups then timeshare with each other, or are processed by different processors. They exchange messages as a form of communication. The effectiveness of the technique depends on the extent to which any system can be divided into these groups whilst still maintaining the high response time and efficient processor utilization offered by the multiprogramming technique.

3 CONCLUSIONS

Up to the present time the problems of controlling communication between the various processes comprising a real-time system have provoked many different approaches. There has yet to emerge a technique which compares favorably with all the items given in the check list in Section 2. A mixture of techniques is frequently used in order to obtain the advantages of each method on those parts of the program where it is most effective. Owing to the diverse requirements of data communication within a system it is probable that considerable advances in the hardware of computers will have to take place before a satisfactory general solution is found. The most promising techniques are the message systems. Hardware improvements required are the ability to allocate read or write access of individual message storage areas to processes so that the dangers of over-writing other messages can be eliminated. Preferably the message pointer manipulation processes should also be performed by special hardware so that time and space overheads can be reduced.

For status data the message system is unlikely to ever be efficient. Here the most reliable technique would appear to be the use of data area I/O routines to administer each area. These I/O routines will use semaphores to prevent microinteractions on the data. Although providing a wide variety of I/O services to the other processes, the routines should be fast running so that the maximum amount of parallel processing is possible.

Developments in languages, operating systems and hardware should provide message communication facilities of a sufficiently general nature to protect the user from many of his present communication problems but there seems no current possibility that his interactions with status data can be equivalently eased.

ACKNOWLEDGEMENTS

The author is indebted to Dr C S E Phillips, Mr K Jackson and Dr R J W Kershaw for the many interesting discussions which helped to formulate this paper.

REFERENCES

1 E W Dijkstra
Co - operating Sequential Processes.
Programming languages. F Genuys Academic Press

2 Habermann
Prevention of System Deadlocks
Communications ACM Vol 12 No 7 July 1969.

3 Institute for Computer Research
University of Chicago
Quarterly Report No 19 Nov. 1968.

RRE Malvern
/3/71

REAL-TIME PROCESS

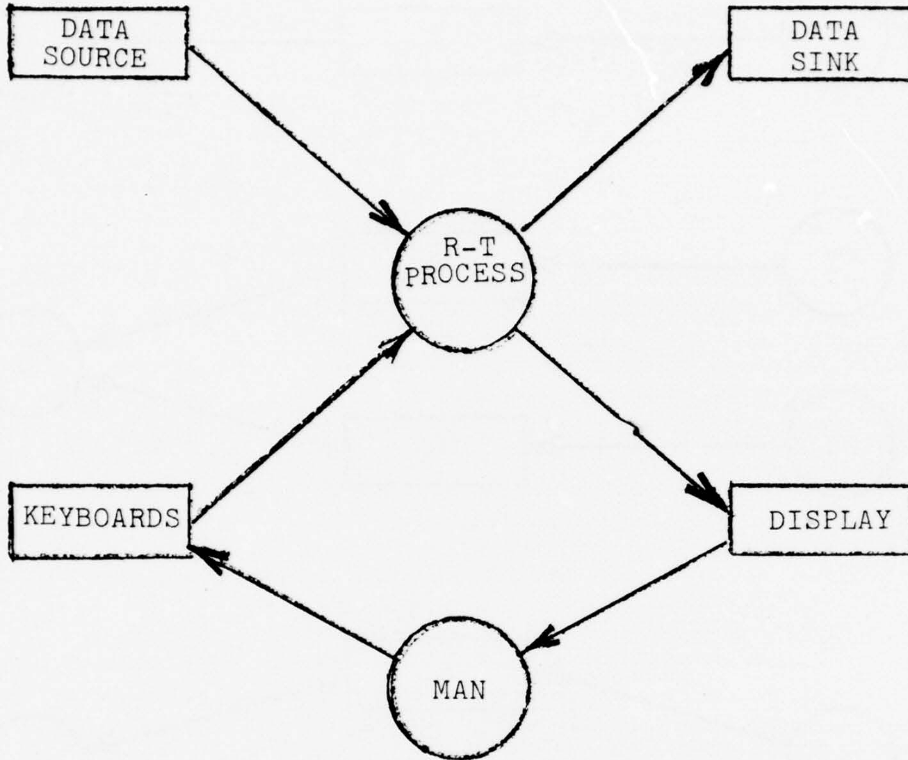


FIGURE 1

TYPICAL MESSAGE INPUT PROCESSING

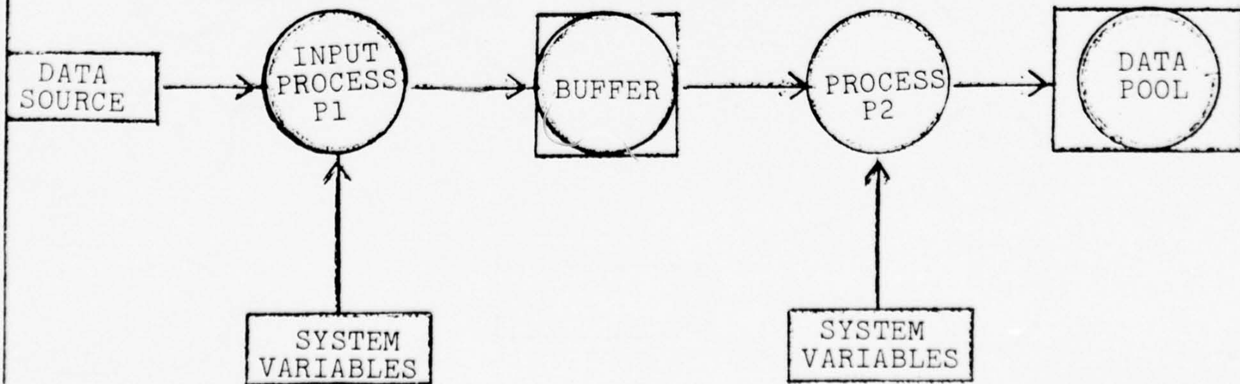


FIGURE 2

PRECEDING PAGE BLANK-NOT FILMED

POSSIBLE INTERACTION BETWEEN PROCESSES

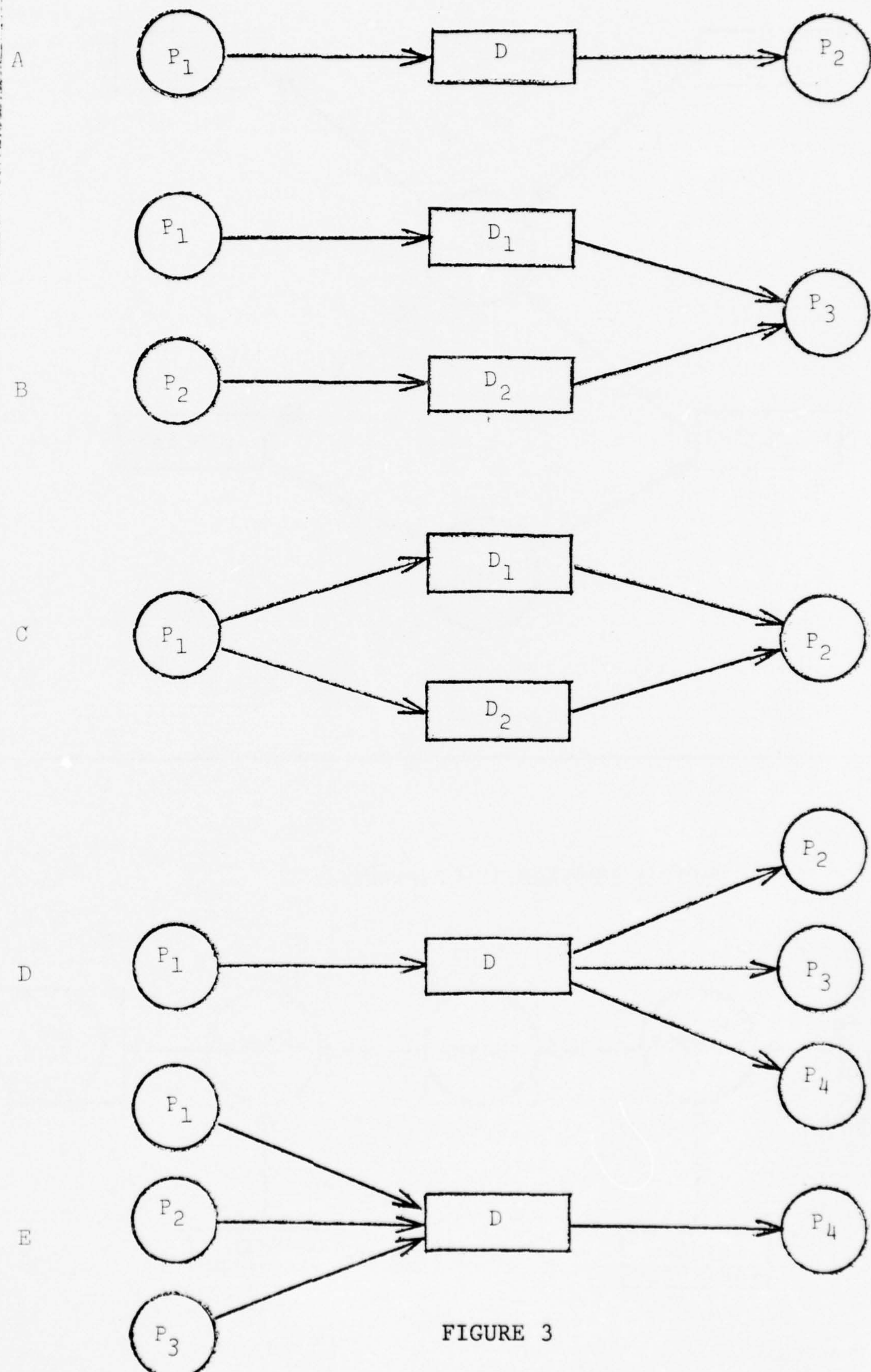


FIGURE 3

DATA BASE I /O ROUTINES

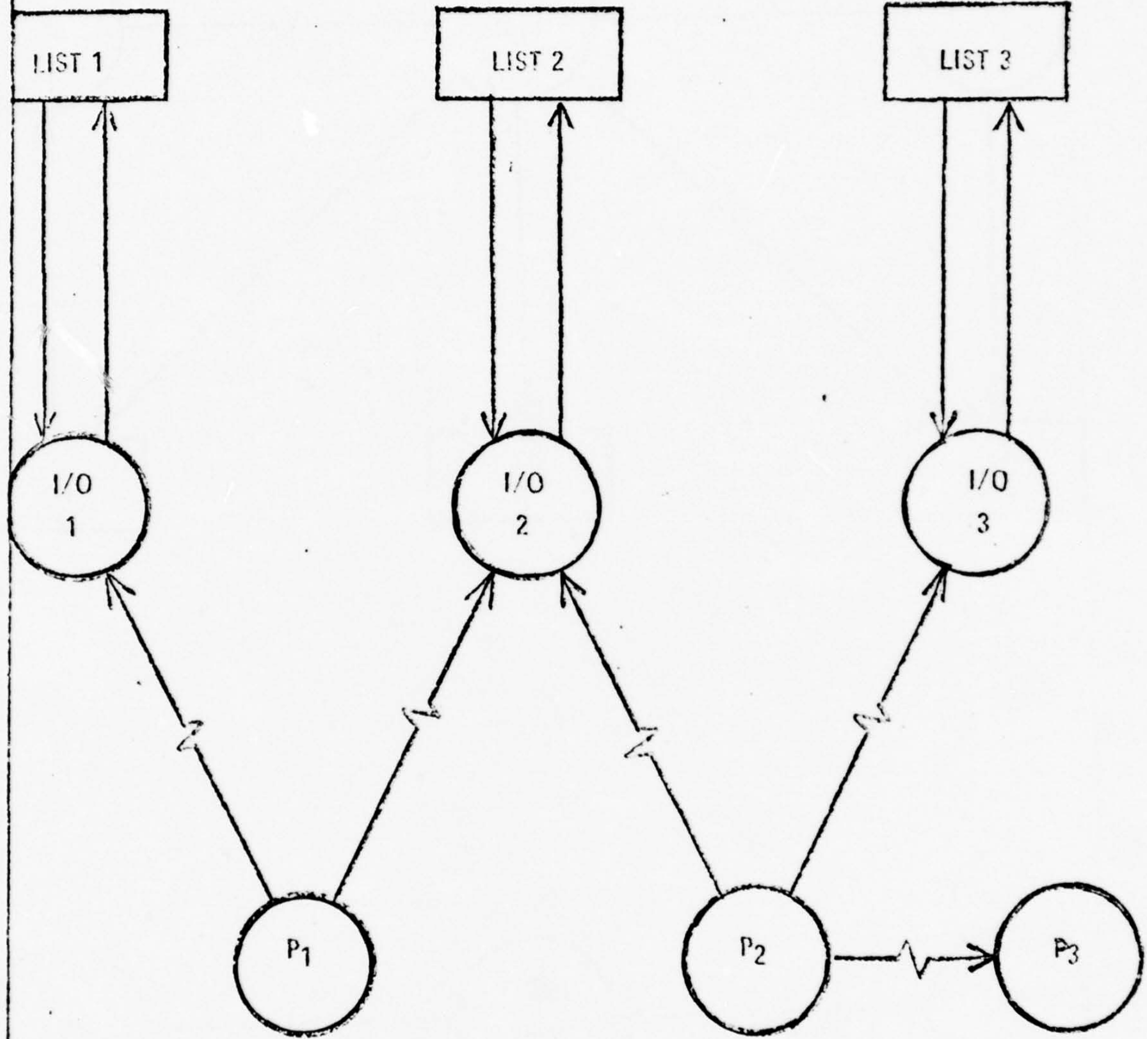
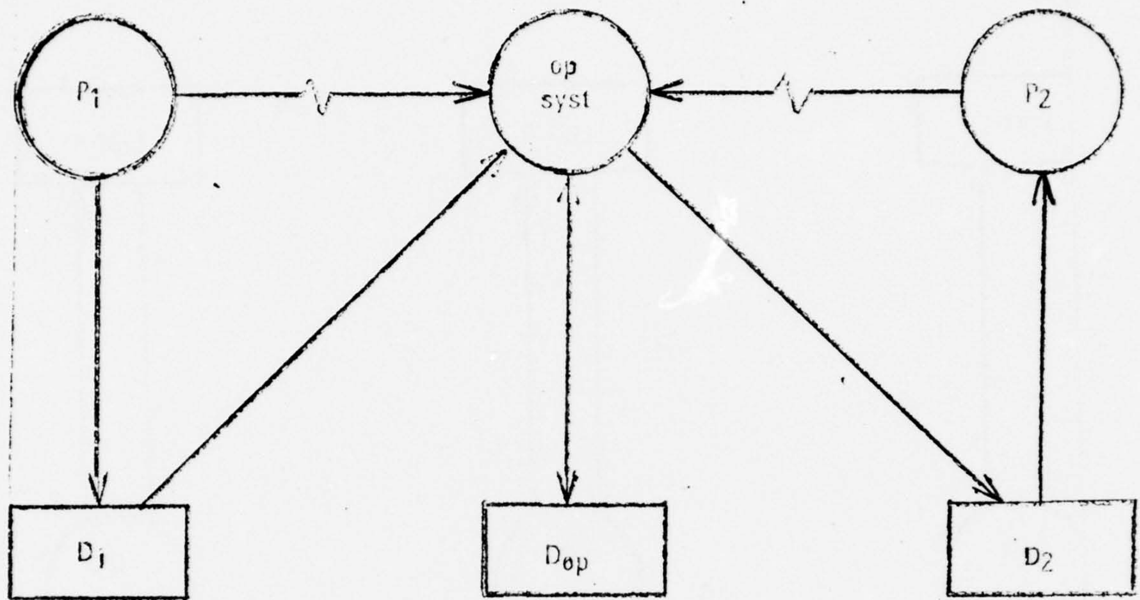


FIGURE 4

MESSAGE SYSTEMS



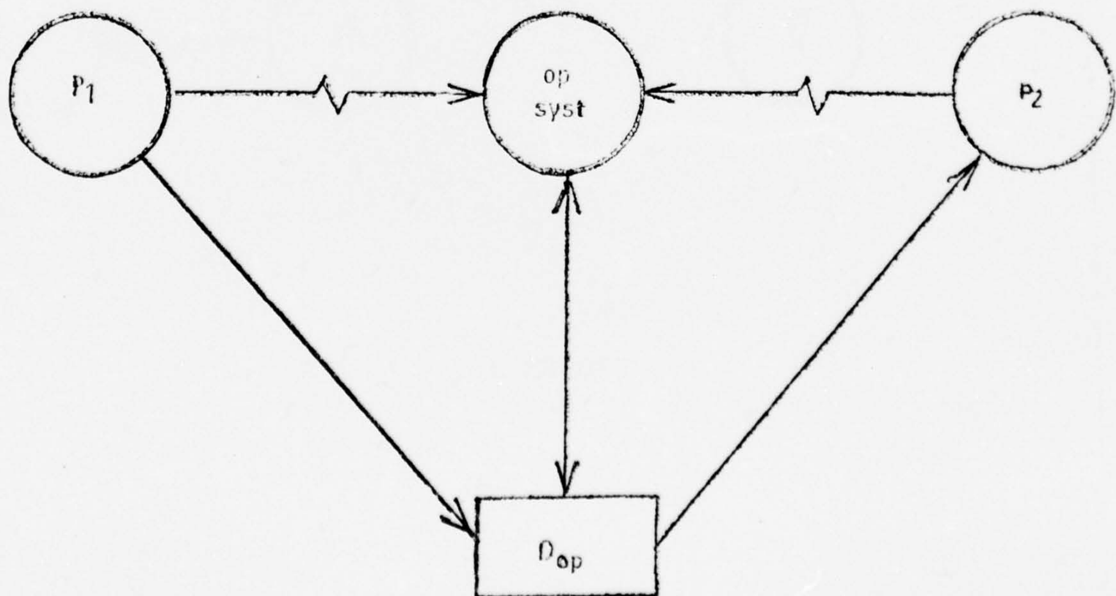


FIGURE 5

MESSAGE QUEUES

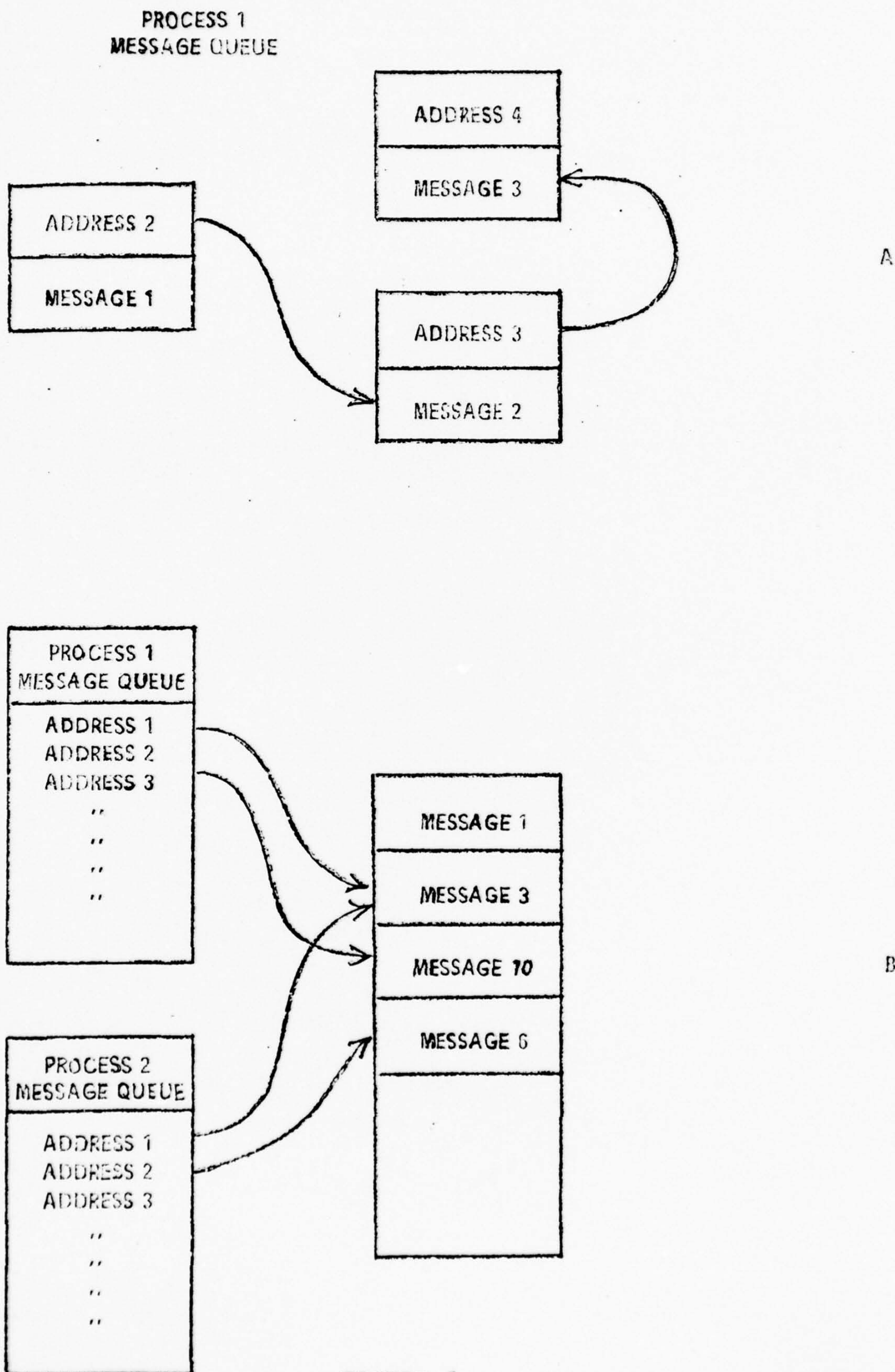


FIGURE 6

APPENDIX 2

A.2.1 SCOPE

This contains a summary of work done at various European meetings, as an attempt to make a formal analysis of the task facilities in four sample languages.

A2.2 TASK STATES

The possible task states may be summarised as:

EXISTENT - the defining code of the task has been made known to the system and all housekeeping done to allow activation. In particular a name has been assigned.

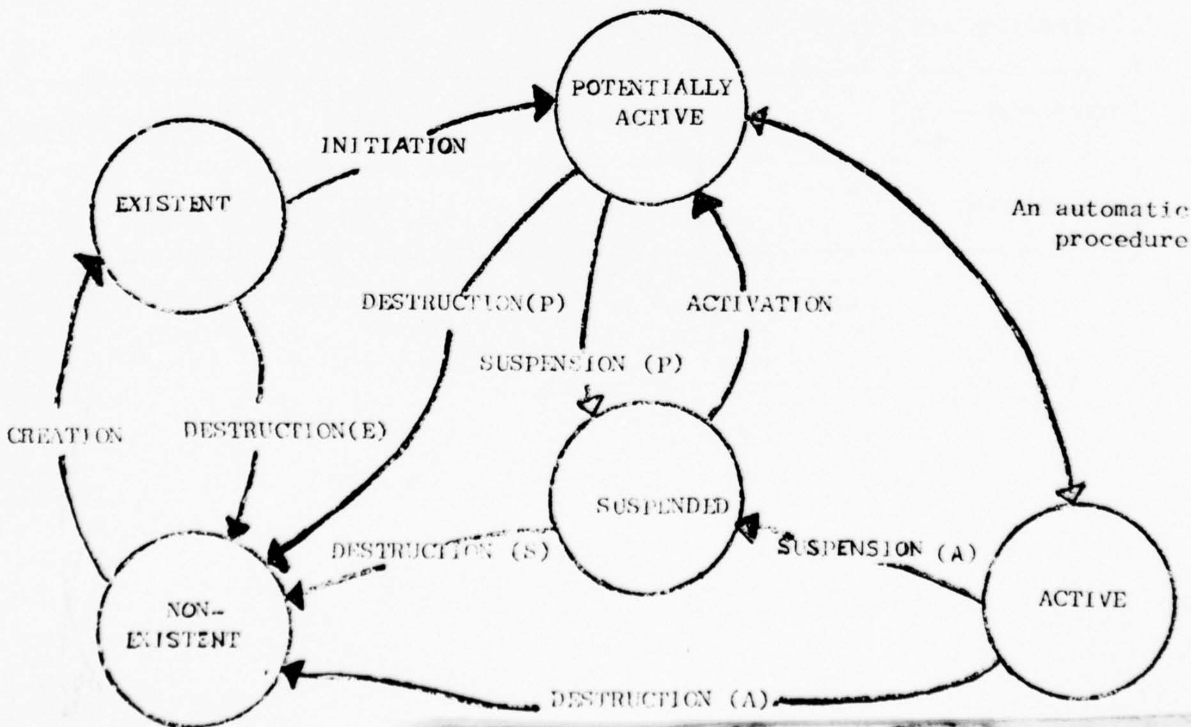
POTENTIALLY ACTIVE - the task has been activated, and is waiting for central processor or other resources before continuing.

ACTIVE - the task has been activated and has use of central processor at this time.

SUSPENDED - the task is held up awaiting some condition (the availability of a resource, or elapsed time).

NON EXISTENT - the name is disassociated from the task, all local workspace is collapsed, and the core made available to other tasks.

Tasking statements have been provided in each of the languages to allow tasks to change their own (or other tasks') states. Schematically these may be summarised as:



PRECEDING PAGE BLANK-NOT FILMED

The transfer between ACTIVE and POTENTIALLY ACTIVE is implemented, under priority control, in all 4 languages/systems. The states are nevertheless distinct (SUSPENSION (P) and SUSPENSION (A) are not the same).

Facilities provided by each of the languages can be briefly summarised as:

State path	LAI	PEARL	RTS	SINTRAN
CREATION	-	DECLARE ACTIVATE	MAKETASK	by loader START
DESTRUCTION (E)	-	-	KILL	ABORT
INITIATION	name (....)	ACTIVATE	START	START,SET,ABSET
SUSPENSION (A)	WAIT RESERVE Auto	DELAY,WAIT REQUEST	DELAY,WAIT etc. STOP SECURE	Auto. PAUSE
DESTRUCTION (A)	RETURN	END of block of code	KILL RETURN END	END, ABORT
ACTIVATION	GO, FREE "timeout"	CONTINUE RELEASE "timeout"	START RE-LEASE STIM etc. "timeout"	Auto. CONTINUE
SUSPENSION (P)	STOP	DELAY	STOP	Auto?
DESTRUCTION (S)	-	TERMINATE	KILL	ABORT,DESTROY
DESTRUCTION (P)	-	TERMINATE	KILL	ABORT

The following table gives the command structure:

ACTION	LAI	PEARL	RTS	SINTRAN
CREATE/DESTROY	Name of procedure (arguments)priority NAME(arg...)PRIORITY Create and start are performed in one operation. Cannot KILL, but self destroy only RETURN	DECLARE taskname TASK Cannot Kill at declaration level	via procedures using SVC's: MAKETASK(prog stack,priority, bufferlength), needs to be STARTED. KILL (stack) (stack task name) Self destroy on completion	via linking loader operations taskname (priority) (Above sets up attribute list without making it known to SV) both CREATE and DESTROY need to be "started"
START/STOP	Start at creation STOP GO	ACTIVATION (taskname) PREVENT DELAY (abs.time internal) CONTINUE(task (taskname) TERMINATE (taskname)	START(stack) STOP (stack) DELAY(interval) RESTART(stack) AWAIT(hours, mins, secs)	START (taskname) SET (taskname, delay) ABSET (taskname, time of day) INTV (taskname, times) (Above makes task known to SV) ABORT (taskname)
COMMUNICATIONS/ MESSAGE	WAIT(logical exp^n of event variables and time expressions) RESERVE (sema) FREE (sema) Global variables Params of procs	WAIT(logical exp^n of event variables) REQUEST (sema) RELEASE (sema)	STIM(S0); IDLE gives (taskname) result; WAIT(S; timeout); SEND(S0); LISTEN() gives stack result; REPLY(); ATTEND (timeout); OPEN(); GET(); PUT(); CLOSE(); SECURE(S,t.o.); RELEASE(S).	COMMON) areas) Global) vari-) ables) Interaction prevented by temp. raising priority. INHIBIT (prior) FREE:

The links between the states shown in the state diagram are not however identical in all 4 languages - the situation is rather complex, but tasking statements can be grouped as follows:

- CREATION

A task can create another task which previously did not exist.
This task is not automatically potentially active: Pearl - DECLARE
RTS - MAKETASK
Sintran - by linkage loader
This task is automatically potentially active: LAI - name (. ..)

Establishment (creation) might usefully be generalised to allow waiting for an event (in addition to time) and possible even messages. A distinction should be made between creating a task which is also simultaneously made potentially active, and creating a task which is suspended, needing a separate activation statement.

- INITIATION

A task can initiate another existent (but not suspended)program:

Pearl - ACTIVATE
RTS - START
Sintran - START, SET, ABSET

- ACTIVATION

A task can activate itself when suspended : All by timeout after suspe[illegible]
or a delayed "SECURE ([illegible]
(or IDLE, WAIT, etc. RTS)

A task can activate other tasks when suspended . LAI - GO
Pearl - CONTINUE
RTS - START/RESTART
Sintran - Automatic?

A task can activate events : Pearl - SIGNAL
RTS - STIM

A task can release semaphores (and hence other programs): LAI - FREE
Pearl - RELEASE
RTS - RELEASE

- SUSPENSION

A task can

suspend until	an event occurs	a semaphore is free	a time	for ever
suspend other tasks			Pearl - DELAY RTS - DELAY AWAIT	Pearl - TERM* RTS - STOP Sintran - ABORT
suspend itself	LAI - WAIT Pearl - WAIT RTS - IDLE Sintran - PAUSE	LAI - RESERVE Pearl - REQUEST RTS - SECURE	Pearl - DELAY RTS - DELAY WAIT	LAI - STOP Pearl - TERM* RTS - STOP Sintran - ABORT

*TERM. = TERMINATE

- DESTRUCTION

A task can destroy

another created, but not yet activated task	RTS - KILL
another suspended task	RTS - KILL
itself	LAI - RETURN Pearl - End of block? RTS - KILL RETURN END Sintran - END
another active task	RTS - KILL Sintran - ABORT

A.2 .3 TASK ATTRIBUTES

The following attribute s are specified implicitly or explicitly (as parameters) by task control statements in each of the 4 languages studied.

		IAL	PEARL	RTS/2	SINTRAN
- Priority		Option, changeable	Option, changeable	Change- able	Change- able
- Name		✓	✓	✓	✓
- Defining code		✓	✓ (1)	✓	✓
- Workspace (local data)	Storage allocated	Implied	Implied	✓	Implied change able
- State space (the contents of which define the state of the task at any instant	Storage allocated	Implied	Implied	Implied	Implied
- Communication space	Storage allocated	✓	Implied	✓	✓
- Code space	Storage allocated	Implied	Implied	Implied	Implied change able
- Monitoring state		Implied	Implied	Change- able	
- Timing (e.g. start/finishing time, repetition frequency)			✓ (2)	Implied Change- able	Change- able
- Device and file allocation			✓		
- Processor identity			Option		
- Name environment (names which the task can access)			Implied	Implied	
- Arguments/parameters		Option	Option		
- Error recovery action or estimation			✓	✓	

(1) The code of a task cannot be changed between termination and the next activation

(2) The timing of a " " " " " " " " "

Informations-, Daten- und Automationssysteme GmbH IDAS

D. N. Shorter

K. H. Timmesfeld

LTPL-European Group

LTPL-E/151

Category: T

Updates: None

Replaces: 107 and 109

pp: 20

Date: 18th January 1974

JOINT

TASKING PROPOSALS

for a LTPL

D 6250 Limburg/Lahn · Kornmarkt 9 · Telefon (06431) 22575

Introduction

This paper is the result of a discussion between D.N. Shorter and K.H. Timmesfeld on the 20. LTPL-E session in London about their respective papers (LPTL-E/107 and LPTL-E/109). It shaws the agreement reached during this discussion. The proposals are numbered according to the tasking comparative study (LTPL-E/105). Some reasons for the choice of each of the proposals are indicated. As with the brevious papers on the same topic emphasis is placed on features rather than syntax.

General Questions:

1.1 Question:
Is tasking included in the language?

Proposal:
Tasking should be included in the language.

Reasons:
Tasking is an essential feature of real time operations. The inclusion of tasking in the language makes programs:

- more easily transferable between different computers,
- more easily understood by programmers using the LTPL for different computers.

Tasking features are too important to be handled purely by subroutines.

1.2 Question:
What is the minimum unit for compilation?

Proposal:
The minimum unit for compilation called a module consists of PA, procedure and global data declarations as well as

PRECEDING PAGE BLANK-NOT FILMED

possible environment definitions.

Reasons:
Separate compilation of parts of a program system allowes modular construction and expansion of such a system (e.g. see FORTRAN and PL/I).

Attributes of PCE's, PA's and happenings

2.1 Question:
What segments (or collection of segments) can be parallel code elements (PCE)?

Proposal:
A segment should consist of any executable statement or collection of executable statements, together with any necessary declaration and initialisation for variable (s) occuring within the statement (s).

Reasons:
General cleanness of language; there is no reason to forbid certain statements to be executed within a PA.

2.2 Question:
Are they (the PCE's) defined explicitly?

Proposal:
Named PCE's are not strongly supported nor attacked. The question is therefore left open for the moment.

Reasons:
Named PCE's might be helpful for diagnostic messages since in such a case it would be possible to print the name of the PA in which the error occurred, the name of the PCE which was executed as the segment of that PA and possibly the name of the procedure which was called from within that PCE. On the other hand, the

association between a PA and a PCE will be static in most cases so that the name of the PA in which the error occurred will be sufficient to locate the error. When this is true, it seems not senseful to introduce a new named item into the language.

2.3 Question:

What are the possible states of a PA visible in the language?

Proposal:

Not existent, dormant, idle, scheduled, runnable, suspended.

Reasons:

This appears to be the most primitive set. The state "dormant" has been included since its existence allows the dynamical attachment of different PCE's to a certain PA which gives more flexibility in programming and will save some storage space in most implementations. This state also coresponds to the proposals of Peter Lindes (see "Minutes of 6th Workshop"). On the other hand a separation of the activation of a PA from attaching a PCE to it allows a more secure handling of the influence of hierarchy on PA-execution (see also 2.6.2, 2.20 and 2.21).

2.4 Question:

What are the visible (and accessible) attributes of a PA?

Proposal:

Its name, its priority and its residence.

Reasons:

A type (hardware or software) as in PROCOL has not been included since the difference between a hardware

and software task is implementation dependent. To ensure a quick execution of a PA it seems more reasonable to give it a high priority and to make it resident. The question of a deadline for a task has been discussed but excluded for the moment since it seems to be too difficult to implement.

2.5 Question:
Does a PA always have a name?

Proposal:
Yes, always.

Reasons:
A name is needed to identify a PA in task operations.

2.5.1 Question:
How is it declared?

Proposal:
A PA's name should be explicitly declared.

Reasons:
All named elements of the language should be explicitly declared.

2.5.2 Question:
If "explicit" give naming sequence.

Proposal:
The question of syntactic forms has been delayed to a later point in time.

2.6 Question:
Can a PCE be identified separately from a PA?

Proposal:

Yes, a logical distinction can be made in any case since a PCE is thecode attached to a PA. Whether a formal i.e. a distinction by name can be made depends from the final answer to question 2.2.

2.6.1 Question:

Can it be renamed, and by what mechanism?

Proposal:

This also depends from the final answer to question 2.2.

Reasons:

When a PCE is not named there is no sense in renaming it.

2.6.2 Question:

Is the association between a PCE and a PA static or dynamic?

Proposal:

There should exist two kinds of PA's, one in which the association between the PA and the PCE is static i.e. given in the declaration of the PA and one in which this association is dynamic (e. g. indicated in the activation of the PA like in PL/1).

Reasons:

The question of the static or dynamic association between a PA and a PCE is strongly related to the block structure of a possible LTPL and the embedding of a PCE into it. When a PCE is embeded in the block structure it may make use of all variables declared in surrounding blocks. In order to enable a simple storage management i.e. a stack for a PA for the LTPL to be developed this has the consequence that the PA associated with this PCE must

not be active longer than the livetime of the innermost of these surrounding blocks from which the PCE is making use of a variable. This can be accomplished either by terminating that PA when the lifetime of this block ceases or by automatically putting the PA executing this block as part of its PCE into a waiting state when the execution reaches the end of this block and the PA in consideration is still active. The most general and sufficiently secure solution would be to make the association between a PA and a PCE completely dynamic on one hand and on the other hand to put the PA executing blocks surrounding this PCE as part of its PCE into a waiting state when it reaches the end of the innermost block out of which the embeded PCE is using a variable and when the PA associated with this PCE is still active. But since this solution is rather difficult to understand a proposal is made to introduce both a static and a dynamic association between a PCE and a PA and to solve the synchronization problem for common variables by putting the PA executing the direct surrounding block as part of its PCE into a waiting state when execution reaches the end of this block and the PA executing the embeded PCE is still active (see also 2.3).

2.6.3 Question:

How is this association made?

Proposal:

In the declaration of the PA for the static association and in the activation for the dynamic association.

2.7 Question:

Does a PA have a priority?

Proposal:

Yes.

Reasons:

Other ways of assigning processor time and/or high level storage seem to be too complicated for the time being.

2.7.1 Question:

How is it defined or assigned?

Proposal:

The priority is given in the declaration of a PA, but it can be changed during the execution of the PA.

2.7.2 Question:

Is it absolute or relative to that of the PA that assigns it, or both?

Proposal:

Absolute priorities seem to be sufficient for most to-day applications, but relative priorities would be a convenient mean for programming several complex problems on one computer.

Reasons:

For to-day process control computers absolute priorities are easy to implement and sufficient for most applications. The question of relative priorities will have to be discussed in more detail.

2.7.3 Question:

Can a PA change its priority?

Proposal:

Yes.

Reasons:

Flexibility in assigning resources to PA's.

2.7.4 Question:

Can a PA interrogate its priority?

Proposal:

No.

Reasons:

As in all four languages compared.

2.8. Question:

What is the strategy of assigning resources to a PA?

Proposal:

Processor time should be assigned to a task strictly according to its priority. The same strategy should be followed when assigning high level storage except when a task is explicitly made resident in this storage. Peripher devices should be assigned to a task only for the time thi task is making an access to the peripheral device in questi Streamlike I/O-devices like printer, plotter, keyboard-input etc. could be assigned to a task for the lifetime of this task. All other reservations could be done by appropriate use of synchronizer variables like semaphores e

Reasons:

The general aim of all such strategies should be to free a resource as early as possible.

2.9. Question:

When are resources allocated to a PA?

Proposal:

High level storage should be assigned to a task which is declared resident in that storage at the time the declaration is evaluated. In all other cases the task should ask for the desired resource at run time. When the resource is free it should be given to that task requiring it which has the highest priority.

Reasons:

The strategy of assigning a resource to a task only when needed and to free it as early as possible is the best for an economic use of this resource.

2.10. Question:

Is it possible to guarantee a PA's resource requirements (e. g. a) core residence, b) processor time) and how?

Proposal:

It should be possible to guarantee a PA's core residence by declaring it resident. All other resource requirements should be handled by the operating system at run time.

Reasons:

See above

2.11 Question:

Can the states of resources be investigated and by what means?

Proposal:

Only by synchronizer variables with ELSE options. If semaphore-like devices are being used it could be useful to have a procedure for determining the value of the semaphore (i.e. the queue length).

Since:

Synchronizer variables seem a sufficient and elegant solution. Queue interrogation would be useful for diagnostic and monitoring routines.

2.12.1 Question:

Can resources be allocated explicitly to a PA by a programmer? If "Yes" than which can be allocated (e. g. working space, processor etc.)?

Proposal:

Except working space for resident tasks no other resource can be allocated explicitly to a PA by the programmer. Of course, it is possible for all other resources to synchronize by use of semaphores or similar variables the time when the actual requirement of this resource from the operating system is made.

Reasons:

As indicated in 2.10.

2.12.2 Question:

Give examples of such allocation!

Proposal:

The syntactic form of such allocation should be fixed at a later stage of the development of an LTPL.

Reasons:

First the semantic form of the LTPL should be agreed upon.

2.13 Question:

Can a PCE be recommenced prior to its completion, and if so, how?

Proposal:

A PCE cannot be recommenced prior to its completion.

Reasons:

Since a PCE is not named by its own, but only the code attached to the named item PA, a PCE cannot be recommenced independently from the PA to which it is attached. A PA however can never be recommenced prior to its completion, since the possibility of a PA being recommenced prior to its completion would mean, that there are two PAs with the same name. This would have the consequence that it is not possible to influence these two PAs separately.

2.14. <u>Question</u>:

Is there only one tasking mechanism - e. g. is there only one way to initiate a PA?

<u>Proposal</u>:

Yes, there is only one tasking mechanism.

<u>Reasons</u>:

This is a matter of conceptual clarity.
Fast response to certain free happenings is a question of the implementation of certain high priority level.

2.15. <u>Question</u>:

Is there a language difference between the declaration/execution of a PCE and the declaration/call of a procedure? Please give examples of all four cases.

<u>Proposal</u>:

As indicated in 2.1. and 2.2. any executable statement could be a PCE. Since according to this proposal a PCE is not named but only recognizable as the code associated with a PA a PCE and a procedure look differently. Furthermore the activation of a PA should have a different syntactic form as the call of a procedure. Examples of the four cases are not given here because the syntactic form of the LTPL should be fixed at a later stage.

<u>Reasons</u>:

The reasons for the difference between a PCE and a procedure declaration are given in 2.2.

A PA activation should not be indicated only as an option of a procedure call, since a procedure call and a PA activation are two completely different things. On the other hand it might be useful to have such a form for the PA activation statement, that it is possible to translate it into the call of an activation procedure.

2.16 Question:

What types of happenings are distinguished in the language and how is each defined?

Proposal:

There should be three main types of happenings:

1. free happenings, e. g. external hardware interrupts and software interrupts,

2. lose happenings, e. g. hardware interrupts arrising from the execution of certain operations (e. g. overflow interrupt as result of a floating point multiplication or addition) or similar software alarms (e. g. alarm when an error in a format conversion occurs),

3. connected happenings, e. g. synchronizer variables like semaphores or other similar variables;

two kinds of connected happenings seem to be reasonable:

a) simple kinds of synchronizer variables, i. e. semaphore variables,

b) more elaborate kinds of synchronizer variables like i. e. the bolt variable in PEARL, which is especially well suited for the administration of data pools common to several PAs.

Reasons:

The distinction between the three types of happenings is made in the following way:

1. Free happenings are happenings occuring unrelated to the execution of any PA.
2. Lose happenings are happenings occuring as side effect of the execution of certain operations of a specific PA.
3. Connected happenings are the effect of synchronizing operations of one PA on a second PA.

2.17. Question:

Please give examples of all linkings of happenings with the execution of PCEs.

Proposal:

Free happenings should more properly be linked with PAs rather than PCEs similarly to time scheduling a PA. Lose happenings should be handled by code within the PCE that indirectly generated them, or defaulted to the executive for standard system action (compare ON conditions in PL/1). Connected happenings i. e. synchronizer variables must be explicitly handled in the code executed as a PA by the execution of synchronizer operations.

Reasons:

Free happenings and lose happenings must be connected with a PA or with certain operations within the code of a PA resp. since their occurance leads to a sudden change of flow of control not known to the programmer. In contrary to this the operation on connected happenings leads to effects completely under control of the programmer i. e. the PA executing a request on the connected happening is put into the suspended state when the happening is not "free".

2.18. Question:

Can a user specify a residence for a PA?

Proposal:

A residence for a PA can only be specified in connection with the residence of the PCE assigned to that PA.

Reasons:

The aim of the residence of a PA is a faster execution of that PA. But in order to execute a PA not only the control block of that PA but also its code i. e. the PCE associated with that PA is necessary.

2.19 Question:

Can a user enable/disable free happenings, at least to the extent of controlling interrupt connection to other PAs?

Proposal:

Yes, this should be possible.

Reasons:

The overhead connected with the recognition of an interrupt not interesting for the system should be minimized i. e. it should be possible to block those interrupts as far outside of the system as possible.

2.20. Question:

Can a PA hierarchy exist? - Is it possible for one PA to initiate a second PA either as an independent entity (while the second might continue execution after the first has finished), or as a dependent entity (when the second would automatically be terminated with the first)?

Proposal:

The question is left open for the moment.

Reasons:

Further discussion is necessary to achieve agreement on that point.

2.21. Question:

Is there the same mechanism for handling both independent and dependent PAs?

Proposal:

Yes, apart from the fact that the continuation of the parent-task at specific points in its code will be dependent of the termination of the subtask there should be no difference in mechanism for handling both kinds of PAs.

Reasons:

All things which are equal should be treated equally. As a comment: The precise proposal to the question heavily depends on "the mechanism for handling".

2.22. Question:

When priorities exist, is there a dependency/relationship between the PA hierarchy and PA's priorities?

Proposal:

For absolute priorities - as proposed in the moment (see 2.7.2.) - there is none. For relative priorities - if such are included - the relative priority of a subtask should be relative to that of its immediate parent-task. In such a case also negative relative priorities should be allowed.

Reasons:

The reasons for absolute priorities are obvious. The existence of relative priorities would have the effect that a task and all its subtasks would have only one priority in competition with other parallel tasks.

2.23. Question:

State any limitation on the number of PAs and hierarchical levels.

Proposal:

Such limitations should only be defined in a specific implementation.

Reasons:

This is a question of machine-size and operating system performance.

2.24. Question:

Can a dependent PA issue all PA-operations?

Proposal:

Yes.

Reasons:

As in all three compared languages where a PA hierarchy exists.

SECTION III

INPUT/OUTPUT AND FILE HANDLING CONSIDERATIONS

Input and output functions, regardless of the data source, and associated file handling aspects are treated in these two papers authored by P. Elzer and P. Helleczek of LTPL-E.

1. "Input/Output Including Process Oriented Input/Output", _Minutes of the Sixth Purdue Workshop on Standardization of Industrial Computer Languages_, pp. 240-254.

2. "Some Requirements to an LTPL Concerning I/O and File Handling", _Minutes of the Eighth Purdue Workshop on Standardization of Industrial Computer Languages_, Appendix V-2, pp. 199-202.

PRECEDING PAGE BLANK-NOT FILMED

- LTPLC -

POSITION PAPER

INPUT/OUTPUT, INCLUDING PROCESS
ORIENTED INPUT/OUTPUT

FIRST DRAFT (WITHOUT FILE HANDLING) (REVISED)

P. ELZER, P. HOLLECZEK

AUGUST, OCTOBER 1971

INTRODUCTION

This paper shall give a short survey of the functional requirements for process and non-process I/O necessary for a LTPL and of the solutions proposed in some other languages.

FUNCTIONAL REQUIREMENTS

In a programming language for realtime applications it is necessary to have available:

1. A sufficiently powerful set of I/O statements and formatting possibilities for input and output of numeric and alphanumeric data (In the following called character-I/O) which is on the other hand not too complex and complicated to implement.

2. A possibility to transfer unformatted data e.g. to and from backing store or between computers in a computer network (called binary-I/O).

3. Tools for particularily process-oriented data transfer with special emphasis on:

   3.1. A possibility to service non-standard external devices on language level for which device handlers don't exist (called primitive-I/O).

PRECEDING PAGE BLANK-NOT FILMED

3.2. Inter-system transfers which do not touch the CPU or even the main computer possible.

3.3. For industrial use often calibrated I/O is useful, by which the external representation of a value can be automatically transformed into meaningful physical entities in which the internal representation is expressed.

4. For two classes of I/O which are non-standard now, but will surely be of great importance in the near future, namely graphic and even acoustic I/O,descriptions should be found. They should allow easy handling of frequent applications of these sorts of I/O and even construction of complex applications out of simple language elements.

5. I/O for an industrial computer language should not be too complicated and too bulky to implement, because industrial computers mostly are smaller machines.

6. File handling is also necessary to a certain extent for an industrial computer language, but for reasons of time and space it is not discussed in this paper though there already have been long discussions on this subject in each working group.

So this paper deals only with the way of writing I/O-statements without regard to the nature of their data sources, which may be devices or files.

The following paragraphs shall each give a short summary of I/O in proposed languages and a comparison with the functional requirements.

PL/1 (Refs.: Pollack & Sterling, A. Guide to PL/1 and IBM TR, 25.096, 1969)

PL/1 has a well developed and highly sophisticated package for conventional I/O and file handling.

1) Character I/O:

a) LIST-directed I/O — without formatting, i.e. there is only one standard format (PAGE and

LINE are possible)

b) EDIT-directed I/O usual (FORTRAN) formatted I/O

Formats: F(fixed point), E(floating point),
C(complex), A(alphanumeric),
P(picture), X(blank, column),
R(remote format), replicators are allowed.

c) DATA-directed I/O self-identifying data

d) STRING-directed I/O a data string in core is used as a data source or destination by a GET/PUT statement.

In the program, I/O is written in statement form.

Comment:

For an LTPL this package is much too complex and must be simplified. LIST-directed-I/O can be replaced by EDIT-I/O with the standard format as a default option. DATA-directed-I/O can also be programmed by means of EDIT-I/O and string handling and so it is not necessary to include this feature which is rather complicated to implement.

STRING-directed I/O

PAGE and LINE are allowed in LIST-directed I/O, but not in EDIT-directed-I/O which is slightly inhomogeneous.

The PICTURE-Format does not give a real one-to-one mapping.

2) Binary I/O:

In PL/1 binary I/O is closely related to file-handling.

The respective statement is:

$$\left\{ \begin{matrix} \text{READ} \\ \text{WRITE} \end{matrix} \right\} \text{ FILE (filename) } \left\{ \begin{matrix} \text{INTO} \\ \text{FROM} \end{matrix} \right\} \text{ (place);}$$

The special properties of a transfer are determined by the attributes of the file involved.

(INPUT, OUTPUT, STREAM, RECORD, PRINT; SEQUENTIAL, DIRECT, KEYED; UNBUFFERED, BUFFERED; BACKWARDS; ENVIRONMENT)

Comment:

On one hand the whole file handling system is too complex for a LTPL and on the other important features, like double-buffering are missing.

Regarded as a whole, I/O in PL/1 is typically batch oriented and has only very indirect references to the physical external devices.
To use the keywords GET/PUT for character I/O and READ/WRITE for binary I/O seems not very logical and in fact contradicts the normal way of speaking and thinking about such problems.

RTL/2 (CIRL, technical notes RJL/71/15, 16/71/11 and JGPB/71/9)
The paragraph on RTL/2 has been replaced by the following revised description of I/O facilities in RTL/2 (by R.J. Long):

"The RTL/2 philosophy is to provide I/O and other system dependent facilities through the basic medium of procedures and supervisor calls. This is in line with the objective of defining a language which can be used for writing systems as well as applications-programs. Thus I/O facilities are provided in RTL/2 not as a standard language feature but as instances of the use of the basic procedure mechanism, the I/O procedure bodies themselves being written in RTL/2.

Development of I/O facilities is therefore pursued as a separate activity with the objective of defining a standard set which will suit most purposes, whilst retaining the flexibility (which would be denied if they were syntactical parts of the language) for augmenting or modifying them to meet the requirements of individual systems or new types of I/O device. The description of the facilities must therefore relate to the currently proposed standard set on the understanding that they are independent of the RTL/2 language.

Standard facilities have only been proposed to date on data handling devices (cards, printers, discs etc.). Standard facilities for plant I/O (analogue or digital I/O) will follow later and it is intended that these should use the same basic procedure or supervisor call mechanisms.

The proposed standard I/O facilities are divided into system independent functions which provide a standard interface to any user who requires conventional I/O facilities, and basic functions built into individual operating systems which implement the particular message buffering, I/O priority handling etc. which may be needed.

System independent functions are:

1) CHARACTER FORMATTING PROCEDURES. These provide a simple conventional package with functions such as

   READ - scan an input stream for a decimal number and convert to a real binary number

   WRITE - output a real number in various conventional formats, e.g. 23.456 or 2.3456E01

   TWRITE - output a string

   and so on.

   They rely on a basic CHARIN or CHAROUT procedure (see 3) for a source of characters or a means of disposing of them. They may thus be used for any device or for internal string creation or analysis.

   Standard procedures will be extended to include more comprehensive formatting systems, similar in scope to FORTRAN, for systems where the overheads of a more sophisticated scheme are justified.

2) FILE PACKAGE. This will be available in three standard forms for control of backing store. The simplest supports a small fixed number of files of predetermined size, with files referenced by number, while the more complex provides a large and dynamically variable file structure with files referenced by symbolic names. In all cases essential write protection features are provided with interlocks to ensure that files can be manipulated where necessary in an indivisible fashion.

Basic Functions specific to an Operating System are:

3) DEVICE OR APPLICATION-SPECIFIC PROCEDURES. These are known generically as CHARIN and CHAROUT procedures. They provide or absorb characters for character formatting procedures (1 above). CHARIN or CHAROUT may drive physical devices directly, may drive devices via subsidiary tasks in a buffered fashion appropriate to the operating system, or may operate only on internal strings or messages. They therefore implement the special I/O characteristics suitable to an application, over which a systems programmer often needs detailed control. They are written in RTL/2 but include occasional machine code statements where a device is to be physically operated or a device interrupt is to be handled. CHARIN and CHAROUT procedures will be available for standard devices in any standard operating system, and thus need not concern the normal user.

   Such procedures include, as a special case of buffering, interactive I/O procedures to allow output and related input to be linked indivisibly, e.g. for conversational use of a typewriter.

4) I/O CHANNEL DESCRIPTIONS. These are small data records of standard RTL/2 form, which the user creates, and define the linkage between the character formatting routines and CHARIN and CHAROUT routines, together with user-accessible error recovery features. They may be separated from the main body of a user program and comprise the only machine or system-specific information a user would normally be concerned with."

Comment (by the authors):

Due to the special philosophy of RTL/2 things like I/O are not included in the language. So they are more or less system dependent. In order to avoid too much diversion which could possibily confuse the user, one should at least standardize these proceaure calls.

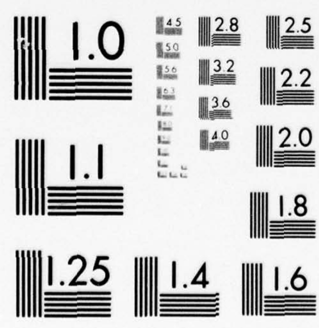
AD-A036 475 PURDUE UNIV LAFAYETTE IND PURDUE LAB FOR APPLIED IND--ETC F/G 9/2
SIGNIFICANT ACCOMPLISHMENTS AND DOCUMENTATION OF THE INTERNATIO--ETC(U)
JAN 77 N00014-76-C-0732

UNCLASSIFIED NL

3 OF 3
AD
A036475

END
DATE
FILMED
3-77

03647



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

L A I (Spécifications techn. d'un language de programmation et d'un moniteur pour les applications industrielles en temps réel, Sept. 1970)

In LAI I/O is divided into two main categories:

1. "Classical" I/O
2. Process I/O

1. "Classical" I/O deals with the I/O of characters and their respective conversions into and/or from machine representation. It is subdivided into two classes:

   1.1. I/O in sequence with a program (the program statement following the I/O statement is not executed before I/O is not finished).

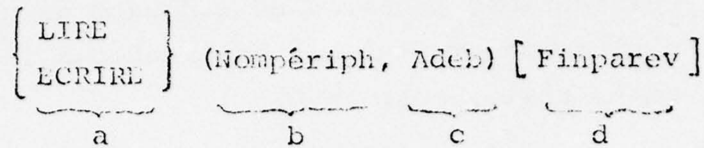
   Statement form:

   $$\underbrace{\begin{Bmatrix} \text{ENTRER} \\ \text{SORTIR} \end{Bmatrix}}_{a}\ (\underbrace{\text{nompériph}}_{b}\ \underbrace{[\text{form}]}_{c}\ )\ \underbrace{\text{Listensor}}_{d}$$

   a) Direction of transfer
   b) Name of peripheral
   c) Label of format which is to be applied on transferred data
      default option: standard format
   d) List of Variables into or out of which the transfer takes place.

   1.2. I/O parallel to the execution of a program (I/O is started as a separate task):

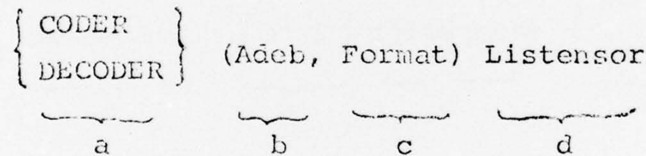
Statement form:



a) Direction of transfer
b) Name of peripheral
c) type of variables involved
d) Event, which can be associated with a transfer for synchronization with the main task.

This type of transfer works in a buffered mode, i.e. information is transferred into or out of a buffer area, from or into which it has to be transported by a decoding or coding operation (CODER, DECODER).

Form of these operations:



a) Type of operation
b) Type of variables involved
c) Format specification
d) List of variables

1.3. Types of formats:

Possible formats within these classes of I/O are:

E: floating - point
F: fixed - point
I: Integer
O: Octal
H: Hexadecimal

"Space", "line" and repetition factors can be specified. Text I/O is also possible.

2. Process I/O is also subdivided into two classes:

2.1. I/O via programmed data link is described by two statements:

$$\left\{\begin{matrix} \text{ENTLP} \\ \text{SORLP} \end{matrix}\right\} \quad \underline{\text{idregperif}},\ \underline{\text{ident}}$$

idregperif describes the data source or destination in the external device and is specified by a declaration of the form:

declarperiflp::= PERIF, nomperiflp, adex, regint, cadreg, typad [,skip] ;

To expand this here in full detail would exceed the framework of a position paper but one can say in short that this declaration describes the following properties of an I/O-device: Symbolic name for device; external address and function code (eventually); internal register, position and length of the information in it; mode of addressing of ext. device; and, as an option, a skip-instruction for the case that the I/O-statement is not executable. Type and length of the internal register is also defined by a declaration.

2.2. Channel I/O is described by one instruction:
TRANS (nomperifcan, listcan);

nomperifcan is the symbolic name of the device and listcan describes a number of functional details of the transferred words, name of buffer area, end of transfer interrupt etc. It is defined by a declaration similar to idregperif.

Comment:

The method proposed for process I/O is indeed as general as possible. But there still remains a certain machine dependence because of the hardware-orientation of the respective statements

Conventional I/O is adequately complete and the subdivision into "sequential" and "parallel" I/O should allow to write rather effective programs. But it can be assumed that a good tasking mechanism easily includes parallel I/O and therefore there is no need for a special class of transfers for this purpose.

PEARL (new proposal, not yet published)

I/O is now divided into two main categories (comparable to LAI):

1) I/O for man-machine communication
2) Inter-system I/O

The first deals with all sorts of I/O where the external representation of I/O was to be immediately understandable by man, while the second category includes all sorts of data-transfer within a computer-system, e.g. binary, primitive I/O.

1.1. Character I/O:

Statements:

READ readdestination [FROM readsource] [WITH characterformat];
WRITE writesource [TO writedestination] [WITH characterformat];

The statement-form has been chosen for sake of readability. readsource and writedestination can be defaulted by standardfiles or standarddevices, characterformat by standardformats.

Format-elements are:

B: binary;
F: fixed-point;
I: Integer;
R: Remote;
E: floating-point;
H: hexadecimal;
O: Octal;
P: Picture;
S: String;
COLUMN, PAGE and LINE.

Character I/O has been adapted to PL/1 to a large extent, except the PICTURE-format. This has been designed for extreme simplicity of use and a one-to-one mapping of format string and output. (Depending on the size of the character-set 3 or 5 control-characters are reserved which can be used literally as character-variables if necessary.)

Source and destination can be either a device or a file.

It is being discussed to replace the conventional R-format by a format string which can be declared elsewhere.

1.2. Graphic I/O:

It cannot be the goal of a real-time-language to include or to replace a complete graphics package but it should provide basic features for the purpose of graphic I/O.

Statements:

SEE graphdestination [FROM graphsource] [WITH graphformat];
DRAW graphsource [TO graphdestination] [WITH graphformat];

The structure of the graphic format is slightly different from the structure of character-format insofar as data and interpretation prescription are completely separated.

Format elements are:

S: scale; O: origin; D: differential;
A: angle; with the modifiers:
X: x-coordinate; Y; [Z];
and:
B: brightness; C: colour; : skip;
M: mode (=point, line [, circle]);

1.3. Acoustic I/O has not yet been defined but is being considered.

2. Inter-system I/O:

There is one statement which describes the data transfer to and from backing store, to and from and in between process devices:

MOVE source TO destination [WITH conversion]

The necessary distinctions between devices for which exist drivers and which are serviced by binary I/O and those which are specific for an installation and therefore have to be programmed in a special manner is done by the compiler according to the definitions in the system division. source and destination can both be devices and therefore it is possible to describe device-device transfers which do not affect the CPU.

conversion describes built-in-functions by which in some cases a fast calibration of data which are read in or outputtet is performed.

Comment:

The whole I/O of PEARL is strongly dependent on the system division which describes devices, data paths, and performs the allocation of symbolic names to them. Without such a system description outside the problem program itself a construction like MOVE would be very difficult, if not impossible. On the other hand a separate system description makes the I/O statements themselves much handier and improves machine independence of the programs.

LTPL (Minutes of the 4th workshop, pp. 134 - 136)

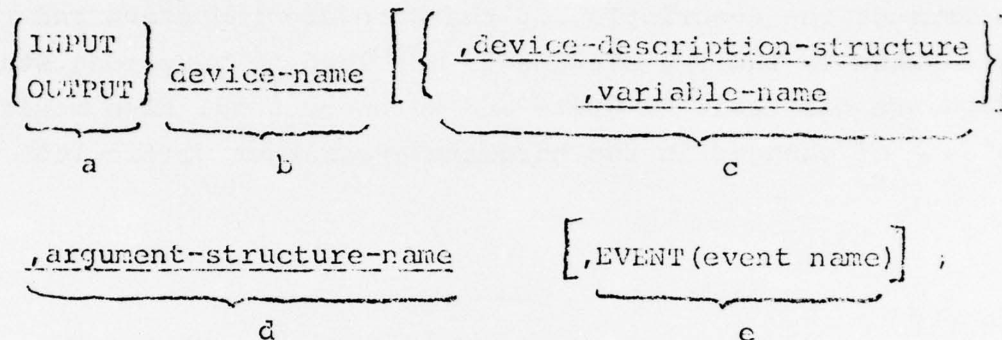
In the LTPL-proposal I/O is very much simplified in comparison to PL/1. It consists of 3 categories:

1) Use of bulk memory.
   The sophisticated file handling system of PL/1 is replaced by a much simpler structure which corresponds to the use of BASED variables.

Keywords available: BULK BASED; ALLOCATE;
FETCH; RELEASE; copy a file.

2) Device dependent I/O (e.g. Process I/O):

Form of a statement:



,argument-structure-name [,EVENT(event name)] ;

d e

a) direction of transfer

b) information which designates the device

c) description of the source of data within the device and of the data path

d) destination (source) of data

e) optional synchronization variable, when an I/O-statement is executed as a parallel process.

Example:
INPUT SCANNER , (5,70,20), A(1), TEMP2;

3) Device independent I/O (External I/O):

This sort of I/O is obviously intended to handle character I/O. Typical statement form:

WRITE logical-device-name, character-string-expression,

character-string-expression determines the number of characters transmitted. Conversions etc. are not done by formats but by built in functions which is made possible by the string-handling capabilities of the language.

Comment:

The idea of separating I/O and formatting helps indeed to improve efficiency and greatly diminishes the size of the I/O package. But on the other hand it might be too difficult for the average applications programmer when he always had to provide for conversion routines.

To include the description of the hardware registers and the data paths in the I/O statements may lead to very long statements which are difficult to write and to understand. Also modifications in case of changes in the hardware are rather difficult.

Summary

So we would propose that the LTPL provides (either directly or by implication) a unique mechanism for the following features:

1) Character I/O like in PEARL (see p. 10, 1.1.), perhaps a little simpler.

2) Basic features for Graphic I/O which allow the construction of Graphic I/O packages should be provided as an option in bigger systems where they are necessary.

3) The description of the hardware structure like in LAI, PEARL and LTPL. It is to be discussed if this description shall be part of the problem program itself or a separate part of the entire program. For reasons of machine independence of the problem program we would prefer to have it as a separate part.

4) On this decision it depends whether process I/O can be one statement like in PEARL or has to be split up into three statements for binary, primitive and device-device transfer.

5) We believe it to be useful to have a hardware description as detailed as in LAI and PEARL. But for further details please see the position paper of G. Mueller (BBC) about hardware-software linkage.

6) The need for user defined error recovery routines is recognized but has not been studied in detail as yet.

7) It must be made sure that the definition of I/O does not lead to an implementation which conflicts with the tasking structure of the user program.

8) Appropriate synchronization mechanisms or statements for the reservation of devices should allow the execution of an output and the respective input response in one indivisible operation.

9) In general I/O for a real-time-language should have some sort of two stage structure: A very basic part for transfer of data without any reference to their format and source. Around this part the more complex I/O functions should be designed.

10) For reasons of efficiency some sort of buffered I/O like in LAI should be facilitated by the language.

LTPL EUROPEAN GROUP
LTPL E/062
Category T
Update 062
Replace 062

August 21, 1972

SOME REQUIREMENTS TO AN LTPL CONCERNING I/O AND FILE HANDLING

P. Elzer

This paper is an updated version of the first outline proposal 062 which was prepared for discussion at the LTPL-E meeting at Mannheim. It was extended according to the results of the discussion and the different items are summarized under the eight criteria which were agreed upon at the Mannheim-meeting for the assessment of any Industrial LTPL whatsoever (c.f. LTPL-E 066), (Minutes of the 11th meeting, p. 3). But the special remarks refer to the present state of PL/I.

1. Completeness

1.1 Extensions for graphic I/O are required because this sort of I/' will play a more and more important part in process control programming (and does already). (c.f LTPL-E-02; Fct. Requ. Thir. of the 6th Purdue Workshop; LTPL-E/044, p. 6).

1.2 Extension for process I/O are necessary, including calibration, conversion checking, programmable scanning, timer setting, resetting etc. (c.f. LTPL-E/018; LTPL-E/02/ Func. requ., Min. of 6th Workshop; LTPL-E/044, p. 6).

1.3 It must be possible to describe direct transfer between hardware-devices or to another computer, e.g., in a CAMAC-system or a PDP-11 environment (c.f. LTPL-E/02; Func. requ. Min. of 6th Workshop).

1.4 It must be possible to describe the structure of real hardware and the data-paths in the real hardware of the system, but this should not be part of the I/O-Statement itself (c.f. LTPL-E/02; LTPL-E/045; LTPL-E/044; Func. requ., Min of the 6th Workshop, LTPL-E/042).

PRECEDING PAGE BLANK-NOT FILMED

1.5 There is a need for elementary, primitive I/O for several reasons: to be able to handle new and non-standard-devices, to provide a two stage I/O structure for better efficiency and extensibility, to allow easy subsetting for small machines. The I/O of PL/I does not contain features which are basic enough (c.f. LTPL-E/02; LTPL-E/045; LTPL-E/044; Func. requ., Min of 6th Workshop).

1.6 It must be possible to create and completely describe data sets within the program and not only by job-control-languages, because a process-control-system does not allow stopping and recompilation of a program for such purposes (c.f. Func. requ., Min of 6th Workshop).

2. Overheads

2.1 Simpler primitives than there are in PL/I are necessary for smaller machines, i.e. without reference to files in I/O in order to reduce the overhead for small system (c.f. 1.5).

2.2 Character I/O has to be simplified, c.g. data-directed I/O has to be deleted in order to reduce the inherent overheads.

2.3 It should be possible to reduce the number of file attributes by deleting those which are not absolutely necessary.

2.4 It should be considered if it is not better to achieve flexibility of programming by simpler primitives which can be composed in a non-standard manner, too, instead of introducing flexibility afterwards by many additional attributes, exceptions etc.

3. Flexibility (at runtime)

3.1 There must be facilities to switch I/O streams between devices in a secure manner. File variables seem not to be sufficient for this purpose. If primitive I/O were used, there should at least be device variables (c.f. LTPL-E/047).

3.2 Formats must be flexible, It is not possible to read in formats or, to compose arbitrary (remote) formats at least the replication factors in formats should be dynamic.

4. Security

4.1 Facilities are required to enable tasks to reserve a peripheral (e.g. during question and answer sequences). It is not clear how PL/I would provide these in a natural manner (c.f. LTPL-E/02; LTPL-E/044).

4.2 File operations should be explicit whenever possible, e.g. implicit OPENing of a file leads to loss of security and can even cause the destroying of data. By explicit operations also the problem with merging attributes and the conflicts between attributes could be avoided.

4.3 There should be clearly defined explicit or implicit synchronization mechanisms for the access of different tasks to the same file (or I/O device). It is not clear how PL/I handles this problem.

5. Robustness

5.1 User defined error recovery routines are necessary. It must be possible to react in a defined manner on e.g. timeout errors. (c.f. Func. requ., Min. of 6th Workshop, LTPL-E/02; LTPL-E/047).

5.2 The program should be able to determine whether an I/O or file-handling operation was performed correctly in order to take alternative actions if required.

6. Predictability

6.1 I/O and file-handling operations should be explicit whenever possible. The whole mechanism should be visible to the user in a defined manner, e.g., distinction between data set and file, definition of the "file-atom" (bit, character, graphic element), creation, opening, reservation of files, structure of the data set, installation of

STANDARD files. Heavy defaulting of file attributes must be avoided.

6.2 Implicit operations and hidden synchronization features can cause strange timing effects. Freak timing, in turn, can cause event handling to fail, causing loss of predictability. Semaphores are more predictable and reliable than events.

6.3 It has turned out that semaphores alone are not quite sufficient. A synchronization mechanism besides semaphores and/or events should be found.

7. Extensibility

The structure of I/O as defined in PL/I gives little scope for extension, but I/O extensibility is needed for new devices and new formats. Extensibility would be aided by more primitive statements, including device handling. One possible route might be to allow the user to define new format functions to be used for new devices. (c.f. Func. Requ., Min. 6th Workshop).

8. Conditioned Responses

Nearly every case of error response should be dealt with in the program and not in the operating system (c.f. 5.1).

P. Elzer

SECTION IV

HARDWARE/SOFTWARE LINKAGES

Two position papers are included on this subject. The first by G. Mueller and B. Girard, Minutes Sixth Purdue Workshop on Standardization of Industrial Computer Languages, pp. 166-183, compares these facilities for the languages PEARL and LAI. The second extends this work to develop a set of functional requirements for this topic. It is authored by G. Louit, Minutes, Seventh Purdue Workshop on Standardization of Industrial Computer Languages, pp. 225-227.

LTPLC Position Paper

NOVEMBER 1971

Hardware / Software Linkage
(Revised)

I. G. MUELLER

(BROWN, BOVERI & CIE, D-68 MANNHEIM, Germany, Abteilung ED)

The PEARL Proposal for the System
Division of a Process Oriented Programming Language

II. B. GIRARD

(CERCI, 22 Rue de Charonne, PARIS II, France)

The LAI Approach

PRECEDING PAGE BLANK-NOT FILMED

The PEARL Proposal for the System Division of a Process Oriented Programming Language

INTRODUCTION

This position paper shows the PEARL proposal to solve the problem of the hardware/software-linkage. This solution, which bases on a partition of the process control program into a "system division" and a "problem division", realizes the Functional Requirements in our opinion in an especially advantageous manner.

A similar solution is proposed by the LAI group. The second part of this position paper shows the LAI approach.

FUNCTIONAL REQUIREMENTS

"Minutes of the Fifth Workshop on Standardization of Industrial Computer Languages", page 83:

"D. Linking Software to Hardware"

Ability to inform the system through a mechanism, such as a symbol table available to the compiler, of:

1. The physical termination identification of each peripheral industrial application component
   (process I/O device) with which it must communicate:
   (sample specification statement)
   Program Identifier = Physical Termination Identifier (ID)

2. The physical identity of the communication channel(s) over which information will be exchanged with each industrial applications component: (sample specification statements)
   Program Identifier = (industrial application component)
   via (communication channel, identifier)
   Data to/from (industrial application component)
   via (communication channel, identifier)

PRECEDING PAGE BLANK-NOT FILMED

3. The physical identification of:

   a. each separable interrupt level and its program identifier;
   b. each separable interrupt point at a given level and its program identifier.

Minutes, page 75:

"b. The requestor can specify an I/O device in a symbolic way. It should not be necessary for the requestor to know a device number and other hardware characteristics peculiar to a device, such as channel, line and equipment numbers".

PEARL Proposal

Syntax: (The following signs of the meta-language have the meaning:

{⊥} :logical "or"
[ ] :optional
••• :repetition)

(1) pearl-program ::= SYSTEM;system-division PROBLEM; problem-division

(2) system division ::= MACHINE;computer-description-list;
DEVICE;device-description;
INTERRUPT;interrupt-description-list;
FLAG;flag-description-list;

(computer-description-list, interrupt-description-list, flag-description-list : discussion not yet finished).

(3) device-description ::=

{ sub-declaration connection - symbol sub-declaration;
⊥
process-terminal-device description; }

(4.1) connection-symbol ::=

{ ←
⊥
↔
⊥
→ }

(4.2) sub-declaration ::=

{ [programmer's-device-name =] [identification] channel-description
⊥
[programmer's - device - name =] identification }

(4.3) process-terminal-device-description ::= process-terminal-device name = [identification] channel - description

(5.1) identification ::= number - in - round - trackets : identifier

(5.2) channel - description ::=

{ * number
⊥
* number . number . number }

SEMANTICS:

(The paragraph numbers refer to the correspondingly numbered syntax rule).

(1): In a very simplified manner one can say, that a process control program has to answer 3 questions:

1st : "where" are the devices, to which information goes, or from which information comes?

2nd : "when" should tasks start or stop?

3rd : "what" should happen, when a task is running?

The system division deals with the first question.
The problem division deals with the second and third question.

That means, the system division has to describe the whole hardware configuration on the language level and therefore connects the tasks of the problem devision both with the process environment and with the non process environment.

The system division contains parts of usual job control languages. It supplies information for the generation of that part of the operating system that is necessary for the special application program and it allows an optimization of resource allocation. It further connects external devices and process interface hardware with symbolic names that are to be used in the problem division.

The last feature allows the programmer "to specify an I/O device in a symbolic way". (Functional Requirements). (See also the LTPLC-Position paper "Input/Output" by P. Elzer and P. Holleczek, University of Erlangen).

(2). By the computer description list the compiler is informed about e. g. the type, model and memory size of the machine used.

The device description specifies the extend and the structure of both the process environment and the non process environment.

In the interrupt description list the programmer can allocate symbolic names to those interrupts he wants to refer to in ON-statements in the problem division. The interrupts themselves are identified by system defined numbers.

In the flag description list the programmer can allocate symbolic names to those flags he wants to refer to in the problem division, e. g. within IF-statements. The flags themselves are identified by system defined numbers.

(3, first rule): In principle, the device description comprises the information, that 2 devices are connected with one another, in such a manner that a flow of data is possible between them. The direction of this flow of data is indicated by the connection symbol (4.1).

(4.2),

(5.1): A device is described by:

1st : a number chosen by the programmer;

2nd : a device designation written out by the device vendor.

The programmer may assign the device any name, to use it in the problem division.

(5.2) : The succession of " * number" describes by which channel of the device the connection was made. If the number is missing, a standard channel was chosen (4.2). If the identifier is missing, the channel number refers to the device name standing above (See e.g. page 8, d, h).

The channel description may consist only of a number when using an analog device. In the case of a digital device, to which several digital process terminal devices may be connected, it is necessary, to specify both the first bit position and the number of bits required.

(4.3) : The process terminal device description means that a process terminal device was connected with a process I/O device. The identifier may be chosen by the programmer. Thereby he can simply formulate process I/O statements in the problem division.

The following example shows how simple it is, to describe the hardware configuration with the language proposed above.

Example:

XYZ0 "CPU1"
XYZ1 "MULTPL"
XYZ2
XYZ3
XYZ3
XYZ4 "DIGØUT"
XYZ5 "TAPE"
XYZ6 "DISK"
20 . . .
65 66 67 . . .
220 . . .
"TEMP5" "PRESS1" "TEMP6" "CØNT1"
"PRESS2"
process terminal devices
process I/Ø non process I/Ø

```
DEVICE;
(a)  CPU1=(1):XYZ0*1<-->MULTPL=(2):XYZ1*3;
(b)               *2<-->        (4):XYZ2*3;
(c)                             (2):XYZ1*0 <--          (31):XYZ3;
(d)                                     *1 <--          (32):XYZ3;
(e)                                     *2 <--DIGØUT =  (33):XYZ4;
(f)                                                     TEMP5   = (31):XYZ3*20;
(g)                                                     PRESS1  = (32):XYZ3*65;
(h)                                                     PRESS2  =          *66;
(i)                                                     TEMP6   =          *67;
(k)                                                     CONT1   = (33):XYZ4*220.1.7
(l)                             (4):XYZ2*4<-->TAPE   =  (51):XYZ5;
(m)                                     *5<-->DISK   =  (52):XYZ6;
```

Explanation of the example:

(a) : programmer's device name (in the following: PDN) "CPU1",
device number (DN) "1",
vendor's device designation (VDD) "XYZO",

device channel (DC) "1", is connected with:
PDN "MULTPL", DN "2", VDD "XYZ1", DC "3".

(b) : DC "2" of the first device above, is connected with:
DN "4", VDD "XYZ2", DC "3";

(c) : DN "2", VDD "XYZ1", DC "O" is connected with:
DN "31", VDD "XYZ3". (Standard DC!).

(d) : DC "1" of the first device of (c), is connected with:
DN "32", VDD "XYZ3".

(e) : DC "2" of the first device of (c), is connected with:
PDN "DIGØUT", DN "33", VDD "XYZ4".

(f) : DN "31", VDD "XYZ3", DC "2O", is connected with:
a process terminal device (in the following: PTD),
which will be addressed in the problem division's
I/O statements by the name "TEMP5".

(g) : DN "32", VDD "XYZ3", DC "65", is connected with:
PTD "PRESS1".

(h) : DC "66" of the first device of (g), is connected with:
PTD "PRESS2";

(i) : DC "67" of the first device of (g), is connected with:
PTD "TEMP6".

(k) : DN "33", VDD "XYZ4", DC "22O", the first bit position
used is "1", the number of bits required is "2", is
connected with: PTD "CØNTI".

(l) : DN "4", VDD "XYZ2", DC "4", is connected with:
PDN "TAPE", DN "51", VDD "XYZ5".

(m) : DC "5" of the first device of (l), is connected with:
PDN "DISK", DN "52", VDD "XYZ6".

Advantages of the PEARL Proposal

The whole hardware configuration is written down in a high level programming language. It is shown that this description also includes the names, the programmer has chosen for the peripheral devices and for the process terminal devices. For this reason the I/O statements of the problem division need only those names, defined in the system division. The I/O position paper of P. Elzer and P. Holleczek shows, how short and easily remembered I/O statements appear.

Using the proposed method it is not necessary to have a separate "job control language". That means that the complete knowledge concerning the hardware/software-linkage of a certain application is included in the program itself. The good legibility leads to considerable savings in the field of documentation.

In principle, the PEARL proposal is already field proven. The process control language PAS1, (developed by BROWN,BOVERI & CIE, Mannheim, Germany), is based on a similar partition into a system division and a problem division. Already very complex process control programs had been written with PAS1, and the practical experience has verified the advantages of the method proposed above.

II. B. GIRARD:

The LAI Approach

With LAI there are two possibilities for I/O processing:

1. STANDARD I/O:

This kind of I/O is character oriented, and dedicated to such classical devices as paper tape reader and punch, line printer etc...

The management of this I/O is obtained by a module of the monitor and the syntax of I/O at a language point of view is very similar to the FORTRAN approach, including READ and WRITE without WAIT. The hardware-software linkage is possible by means of ASSIGN statement.

2. INDUSTRIAL I/O:

In this area the situation is very different. Peripherals are non-standard, without device handlers at a language level, because of an extreme diversity of devices and the possibility of a wide variety of connections between these devices and the CPU.

From a programmer's point of view the requirements are only to have the same mechanism of communication in terms of time constraints, size of data which are transfered, possibly alteration of sequential execution, and identification of sources and destinations of data.

PRECEDING PAGE BLANK-NOT FILMED

From a compiler point of view, much more information is necessary to describe:

A. - The communication channel which is used to exchange information.

B. - Each industrial device which is connected to this channel (number of input output registers, size, address and sub-address of each register part of the device ...).

C. - The mode of transfer possibility of interruption, of skip instruction ...).

This description of peripheral is made separatly from the application itself in a global division which looks like the PEARL's system division.

This is done for machine independence. The objective is to have no reprogramming effort in the case of changing a computer or an application. In such a case, only the global division must be re-written. The independence covers only the computer and peripherals which are connected on the same kind of channels.

Three kinds of communication are considered:

- by priority interrupt;
- by programmed channel, that is to say a channel for which each elementary transfer is under control of an input/output instruction;
- by data channel (cycle stealing).

1.1. Priority interrupt:

Each interrupt source can receive a name. After the name, the priority level (and possibly a subpriority level) is described.

Example: INTRPT ITCAD 5 or
INTRPT ITCAD 5, 3

The connection of an interrupt procedure to an interrupt is made by a CONNECT instruction which arms and enables this priority level.

1.2. Programmed Channel:

In this mode the exchanges are made in rapid succession under the control of an I/O machine statement. The transfer is always made between an internal register accessible by program (generally, the accumulator), and an external one. According to the direction, the internal or external register is the "origin" or the "consignee" of information.

A peripheral is usually constituted with one or several external registers. Some of them are only "origin", others in only "consignee", others sometimes "origin" and "consignee". Some of them contain information itself, others commands, others address extensions of the peripheral itself.

External register addressing mechanism and the methods of exchange on the programmed channel, i.e. internal register writing, are detailed as follows:

1.2.1. Addressing:

There are 4 possibilities for addressing:

A. Direct Addressing (DIR)

The external register has an hardware address. This address is included in the I/O statement. Sometimes, function - bits are distinguished in the address-part (flag-ZERO POINT). It is necessary to define as many registers as there are functions, even though some of them may have the same physical identity.

Example: Read PDP 8 Teletype-register

I/O statement is: 6000_8. On input, we read the TTI register of which the address is 03x:

Input action is precised with defining function-bits in the address which is now, for example, 34 or 36.

The 2 statements address is now: 6034_8 or 6036_8.

The first one makes a logical OR between the contents of AC_{4-11} and TTI register.

The second one begins by ACC-reset, transfers the content of TTI register, flag-reset which will allow hardware to reload TTI register.

We must define 2 peripheral registers: TTI 1 and TTI 2.

The programmer will know the difference by use of the 2 registers.

B. Internal Indexed Addressing (INI)

Often the statement address part is inadequate for addressing all peripherals or the I/O addressing hardware cost is such that addressing possibilities are extended by the use of an internal register, such as the index register.

If the peripheral address is AD and if we want the nth register, a machine internal register will be loaded with n, then I/O AD will be made.

C. External Indexed Addressing (INE)

Similar to paragraph B, but "n" is sent to an external register before I/O-AD. This register can be direct, internal indexed, or external indexed addressed.

D. General Addressing (GEN)

I/O statement adresses a memory word on the more general mode permitted by the computer (indirect, indexed, based).

The addressed word or the obtained address, includes the peripheral address.

This word is sent on the programmed channel before I/O exchange.

Example: On Sigma 2 computer, the required peripheral is in the external register, situated in the P-rack, on the L-line, on the C-controller.

In assembly code:

RD PPERI where denotes indirect addressing.
In PPERI, we have the address of the word:

1.2.2. Programmed Channel Statements and declarations:

Statements:

INPUT : programmed channel input.
OUTPUT: programmed channel output.

The possibility of skipping after testing a peripheral, encountered in certain peripheral, is given in the peripheral declaration.

Example: TTI teletype input external register has been declared.

CAR is the identifier that must receive the value.
INPUT TTI, CAR

or

INPUT TTI, CAR (k + i).

Whichever mode of addressing is used for TTI.

Instead of describing the exact syntax of declarations, some examples follow.

1st Example: READ teletype keyboard information:
- teletype address: 30_8
- Direct Addressing
- 2 kinds of working for the teletype:

1) INPUT statement transfers the teletype-buffer into the accumulator, whether its status FREE or not.

   The programmer must know the peripheral status (generally, an interruption is used for that).

   In this case, the address is 34_8.

2) Exchange is made only if the teletype is free causing a skip. In the other case, INPUT is not completed.
   Peripheral Address: 36_8

Example: Internal Register Declaration

```
REG ACCUM (16)

Peripheral Declaration
PERIF TT11, OCT '34', ACCUM, 4, 8, DIR;
PERIF TT12, OCT '36', ACCUM, 4, 8, DIR, SKIP;


INPUT STATEMENTS
INPUT  TT11, CAR ;
INPUT  TT12, CAR ;
```

the accumulator will be entered in positions 4-12 with the contents of the single external register of whose address is 30_8.

In the first case, the function-bit (4) is used to perform an unconditional exchange.

In the second case, the function - bits are used and allow the transfer only if the peripheral is free.

2nd Example: Contacts - Matrix

We have 20 matrices, each with 16 lines of 16 contacts. Only one line can be read at a time.

1) To get the I-line of the M-matrix, the accumulator is loaded with I and an input-statement is made with the effective address of the M-matrix.

We shall write :

```
PERIF  MAT, HEX '40', ACCUM, O, 16, INI ACCUM;
REG ACCUM (16);


INPUT MAT (M, I), VANNE(k) ;
```

The resulting address is 40+M; ACCUM will be loaded with I.

We can have:

```
INPUT MAT (M, FUNC (I)), VANNE (k);
```

2) 16 bits external register called REX is loaded with I for selecting the line ; we shall have the same declaration for the accumulator internal register.

peripheral declaration

```
PERIF MAT, HEX '40', ACCUM, O, 16, INE REX;
PERIF REX, HEX '30', ACCUM, O, 16, DIR;


   ENTLP MAT (M, I), VANNE (k);
or ENTLP MAT (M, FUNC (I)), VANNE (k);
```

For data channels, a similar approach leads to a convenient mean of communication for INPUT and OUTPUT.

The characteristics of the channels (possibility of interruption, of data chaining, of locking CPU during transfers, etc ...) are described in a declaration.

G. LOUIT
STERIA-FRANCE
March 29th, 1972

LTPL EUROPEAN GROUP
LTPLE/Ø 31

Category T
Updater : LTPLE/ØØ9
Replaces : none
3pp

POSITION PAPER

HARDWARE - SOFTWARE LINKAGE

The Hardware Software linkage problem has been described in LTPLE/ØØ9 for PEARL and LAI and in LTPLE/Ø32C for PROCOL. The fonctionnal requirements has been stated in the 6th Workshop, green printing, page 99. With the help of these examples, it is possible to distinguish in depth functionnal requirements.

1°) General Organization

An application is implemented at two levels :

- a system level
- an application level

The application level is described with the language (LTPL) and expresses all the problems of programming actions and reactions to events.

The system level is not described with the language (LTPL) but with a more dependant (efficiency) machine one ; it expresses relations between tasks (events, interrupts, timers, etc...) for the particuliar application.

The application programme describes the organizational parameters of his probleme to the system level via somme mechanism (conversationnal is easy) e.g. tasks, interrupts, devices, ...

./.

2°) Computer

The interrupt system particular to the computer will be described to the system : levels and sub-levels, number of levels.

Relations between symbolic interrupt system and computer interrupt system will be stated at the system level

Relations between symbolic interrupt system and tasks will be defined at the system level.

Distinction, if suitable, between hardware and software symbolic interrupts.

3°) Classical devices

- Characteristics of the Discs availability (PROCOL) should be described a at the system level.
- Characteristics of the teletypes and other classical peripheral devices s should be described symbolically and physically at the system level. (Buffer size)

  (In PROCOL, for each peripheral, a hardware task assume the input output function).

4°) Industrial peripherals

- The system should be aware of the identification of each module.
- Distinction between types of i/o (analogie, digital, ...) should be made
- Pseudo symbolic names are given to devices at the system level and are assigned by physical characteristics (control unit address, number of charmels, mode of connexion). It could be useful to change dynamically of mock.
- At the application level, the user should be able to designate in a symbolic way the device (terminal plus a list of charmels). (In PROCOL, declarations for Industrial i/o are made in the COMMCN part of a section of compilation).

./.

5°) Failure

Dynamic change of device in case of failure should be possible. For example and aid-device should be specified at the system level. (this is done in Procol for the teletype of the system.

SECTION V

SOME OTHER IMPORTANT CONSIDERATIONS

The remaining topics included in this Part each have only one paper connected with them as a result of the editorial work involved in preparing the Minutes of the individual Workshops and this Summary of those publications. This in no way negates their importance; it is merely an indication of their popularity with the LTPL-C members compared to the topics considered earlier here. For want of any other organization, the papers are presented here in chronological order of their original appearance in the _Minutes of the Purdue Workshop on Standardization of Industrial Computer Languages_.

1. "Data Types and Manipulation", _Sixth Workshop_, pp. 190-193, by R. C. Dennis.
2. "Acceptance Criteria", _Ibid_, pp. 260-264, Anonymous.
3. "Static Program Structure", _Seventh Minutes_, Appendix IV-2, pp. 195-203, by Peter G. H. Rieder.
4. "Extensibility: The Operator Statement", _Ibid_, Appendix IV-10, pp. 235-245, by Konrad F. Kreuter.
5. "Operating System Interface", _Eighth Minutes_, Appendix V-3, pp. 203-208, by G. Verroust.

PRECEDING PAGE BLANK-NOT FILMED

DATA TYPES AND MANIPULATION

R. C. Dennis

Data: Information that is operated on during program execution.

Problem Statement: Development of data facilities for a language which fulfills the functional requirements of industrial computer systems.

Functional Requirements:

1. All data can be initialized.
2. The only implicit data type conversion occurs during assignment or with I/Ø operations.
3. All problem data is represented either as arithmetic or as string data.
4. Arithemtic data are either fixed or float. The attributes of base, scale and precision are implicit within each implementation. Arithmetic constants are arithmetic data items which take as name their value (hence the value is fixed throughout program execution.) Arithmetic variables are arithmetic data items without such constraint.
5. There are two kinds of string data, character strings and bit strings.
6. Character string constants consist of strings of characters from the data character set, enclosed in quotation marks. Character string variable can assume any of a range of values corresponding to character string constants. Maximum string length is implementation dependent. There are fixed and varying length strings.

PRECEDING PAGE BLANK-NOT FILMED

7. Bit strings are similar to character strings except that the elements of the string are the binary digits 0 or 1. Boolean data are bit strings of fixed length 1.
8. Program control data are of the types label, entry, task, event, points, offset and area.
9. Data are organized as:
    a. Scalar Data (single items)
    b. Data Aggregates
        - Arrays (wherein all elements of the array are of the same type.)
        - Structures (all elements do not have to be the same type.)
        - Arrays of Structures
10. Names are either single, or subscripted or qualified. Qualified names are unambiguous structural references.
11. Data Manipulation There are three types of expressions: scalar, array, and structure. All operations performed on arrays or structures are performed on an element by element basis.
12. A scalar expression is any of the following:
    - A constant
    - A scalar variable
    - A function reference
    - Any expression enclosed in parentheses
    - Any expression preceded by a prefix operator
    - Any two expressions connected by an infix operator

A scalar expression returns a scalar value. The type of the value is the type of the expression. The type of the expression is dependent upon the class of operators; i. e., arithmetic, comparison, bit string, or concatenation.

Operator with Data Type of Operand

Scalar Array	Prefix + - Infix + - * ** & | ¬ || Comparison	Fixed Float Bit Character

If there is no operator, the class of the expression depends on the type of the single operand. Only the operators = and ¬= may appear with label, pointer, and offset data. No operators may appear with entry, area, event, and task data.

13. An array expression returns an array value.

14. A structure expression returns a structure value.

15. The comparison operators are:

16. Arithmetic comparisons involve either two fixed operands or two float operands.

17. Character comparisons involve left-to-right, pair-by-pair comparison of characters. If the operands are unequal in length, the shorter is extended to the right with blanks.

18. Bit comparisons are handled in a manner similar to character comparisons, except binary zeros are used to extend the shorter string.

ACCEPTANCE CRITERIA

PROBLEM

To ensure as far as is practicable that the LTPL finally proposed by the Workshop will prove an acceptable and effective standard.

REQUIREMENT

The LTPL Committee's objective is to develop a language which meets the functional requirements of industrial computing. Since existing languages appear not to offer a suitable subset, the Committee will therefore be making proposals for a new language or extensions of an existing one.

It is also the aim of the Committee that the proposed language should become a universal standard for industrial computing.

Experience shows that design deficiencies in a language are costly to live with or rectify, and defeat the purpose of a standard. It is therefore a prime responsibility of the Workshop to ensure, as far as is practicable, that the LTPL is viable before promoting it as a standard. Reasonable evidence must be given that the language meets practical criteria as well as being theoretically sound.

PROPOSED SOLUTION

There is clearly a practical limit to such demonstration, but it would not be out of the question to set out in advance the criteria against which the LTPL proposal will be finally judged by the.Workshop. These should cover both functional requirements and other practical criteria.

The following procedure is proposed:

1 The LTPL Committee's final proposal for a language will be submitted for approval by a two-thirds majority of the Purdue Workshop before any attempt is made to secure standardization.

2 On submission to the Workshop, a written appraisal of the language in the form given in the Appendix to this Working Paper will be provided to each voting member. The appraisal will be agreed and written by the LTPL Committee.

3 This appraisal and a language spedification will be in the hands of Workshop members at least 3 months before vote is taken.

The object is to ensure that the Workshop as a whole supports the proposal on well constituted evidence, and is seen to do so.

APPENDIX : FORM OF APPRAISAL

To all members of the Purdue Workshop

1 LTPL SPECIFICATION

A complete specification of the proposed LTPL is included with this appraisal. This specification is agreed by the LTPL Committee:

(a) unanimously
or (b) with minor dissent on the following items:

....

or (c) with disapproval by a substantial minority on the following general grounds:

....

The Committee would like to see the following extensions to the language but is unable to specify these exactly at the present time:

....

2 APPRAISAL: TECHNICAL CRITERIA

2.1 Semantics

2.1.1 Semantic range

(Author's note: In the final form of this Appendix there will here be a complete list of the LTPL functional requirements suggested for the language in so far as they imply operations which the LTPL should be capable of expressing. They will be grouped under suitable headings e.g. Data Management, Program Control, Task Management, I/O, etc; many can be gleaned from past Workshop and Committee Minutes. In each case it will be required to state whether the operation can be expressed (a) directly or (b) indirectly and, if so, how; examples to be given).

2.1.2 LTPL machine description

(Author's note: In the final form of this Appendix it will here be required to give (or refer to) a definitive description of all operations implied by the LTPL but not directly compiled from it. In particular, it will be required to cover the aspects of task storage allocation, task control and communication, I/O, etc, such as to give an accurate description of the facilities assumed to be provided by the operating system or environment of an LTPL program).

2.1.3 Computer independence

The language

(a) is completely machine independent
or (b) includes the following features which may lead to significant disparity of execution efficiency when applied to different types of computer:

....

(Objective appraisal to be given, especially with respect to data types and structures).

2.2 Syntax

2.2.1 Definition

A complete syntactical description of the language is given in ....

2.2.2 Character set

The syntax implies the following language character set:

....

2.2.3 Format

The syntax implies the following rules for layout and formatting of programs:

....

2.2.4 Appendages

The following language features are provided for insertion of macros, code sequences and comments:

....

2.2.5 Integrity

The following syntactical features have been introduced to aid detection and elimination of program errors at writing and compile time:

....

2.2.6 Modularity

The syntax implies the following modularity features for program management and compilation:

....

2.3 Implementation

2.3.1 Character set

The language character set is available on the following standard peripheral equipment without major modification.

....

2.3.2 Compiling requirements

It is known/estimated that compiling is practicable on the following typical hardware configurations:

....

2.3.3 Program development

The following is a brief description of the stages involved in producing a running program:

....

(Description to cover compilation, link editing, testing etc).

2.3.4 Code efficiency

The following is an appraisal of the efficiency of object code obtainable using the compiling resources described in 2.3.2:

....

(To include estimate of program sizes compared with those produced by assembly coding under similar structural constraints).

2.3.5 Code Integrity

The language design permits the following run-time checks to be included without impairing the above efficiency (see 2.1.5):

....

3 ASSESSMENT: SUPPORTING EVIDENCE

3.1 Existence

3.1.1 Compilers

The following LTPL compilers now exist:

....

(Details to be given of target machines and compiler characteristics).

3.1.2 Running programs

Programs have been compiled and run on the following computers:

....

(Details to be given of assembly methods, operating systems used, nature of code produced etc).

3.2 *Evaluation*

3.2.1 Trial Problems

The following example problems have been programmed in the LTPL:

....

(Problems to be decided in advance if possible; print-out's of program text to be provided).

3.2.2 Conclusions

The following is an appraisal of these evaluation tests:

....

(To cover object code and run-time behaviour if applicable).

3.3 Field trials

3.3.1 Application

The language has been used in the following industrial applications:

....

(Details to be given where possible).

3.3.2 Conclusions

The following is an appraisal of the performance of the language in these applications:

....

(Quoting user experience where possible).

4 ASSESSMENT: GENERAL ACCEPTABILITY

4.1 Standardization

4.1.1 Proposed route

It is proposed to seek standardization through the following procedure and standards organisations:

....

4.1.2 Qualifications

The following difficulties are expected to be encountered in its acceptance as a standard (other than that it is not a standard already):

....

The following general points are in its favour

....

4.2 General Acceptance

4.2.1 Vendors

The following vendor organisations have indicated that they would support adoptance of the LTPL:

....

4.2.2 Users

The following user organisations have indicated that they would support adoptance of the LTPL:

....

Peter G. H. Rieder
Siemens AG, E AUT 43
April 7, 1972

LTPL European Group
LTPLE/14

Category T
updates none
replaces none
8 pages

Position Paper on

S T A T I C   P R O G R A M   S T R U C T U R E

PRECEDING PAGE BLANK-NOT FILMED

Introduction

This paper is intended to continue the work which was started at the 6th Workshop by a listing of functional requirements concerning static program structure. Those requirements, however, merely referred to the question, how a program should be structured statically, i.e. before it is actually running. In the following also some problems concerning the interdependence and appropriate separation between static and dynamic program structure will be treated.

Since in industrial applications run-time restrictions occur very often, in a sense static program structure is a functional requirement itself: As far as possible the structure of a program should be determinable at compile-time. Moreover, all decisions should be feasible statically unless they depend on information necessarily attainable only at run-time.

Functional Requirements

A programming language for real time industrial computers must contain the following features:

1. There are means (e.g. special key-words) to group statement into modules, i.e. units of compilation.
2. It is possible to indicate, which entities in separately compiled modules are identical, i.e. must be linked together.
3. There are lucid scope rules for all entities.
4. Explicit declaration is required where standard defaults or contextual declaration cannot supply reasonable, understandable attributes.
5. There is a possibility to specify inline-code.
6. All procedure parameters must be completely specified; especially with formal procedures also the corresponding higher-level formal parameters are required.

PRECEDING PAGE BLANK-NOT FILMED

Problems

7. How far can storage allocation be prepared at compile time ?

8. Shall the language provide pointers specified by the data type they point to (reference-concept) ?

9. Shall there be several classes of parallel activities (tasks, I/O-Operations, interrupt-responses) ?

10. What can be activated as task ?

11. Shall the task hierarchy, if it exists, be determined at compile-time ?

Solutions of LAI

1. Module delimiters are not part of LAI itself, but belong to a separate job control language.

2. Global declarations form a separate so-called article initiated by the key-word GLOBAL. In other modules references to those global entities are indicated by declaring the corresponding identifier with the attribute EXTERNAL.

3. The scope of entities not explicitly declared global or external is the article containing the corresponding declarations.

4. Except the usual contextual label declaration all entities are declared explicitely.

5. LAI includes a macro facility enabling the programmer to insert instruction sequences on source-code level.

6. All formal parameters must be completely specified. However, procedures can not be formal parameters.

7. Storage allocation is not part of the language itself. However, since the maximum space requirements of all items are constant, core storage management, though left to the supervisor, may be supported by the compiler.

8. LAI does not include pointers.

9. In LAI there are three different kinds of parallel activities: tasks, interrupt-responses, simultaneous I/O - operations.

10. Only a so-called process may be activated as task. A process is different from a procedure and also from the code of an interrupt response.

11. There is no task hierarchy in LAI.

Solutions of PEARL

1. A module is initiated by the key-word MODULE and terminated by the key-word MODEND.

2. Entities common to different modules must be declared in each of these modules using the attribute GLOBAL. On linkage identification of those entities works by their name and an additional qualifier specified after GLOBAL; thus within a module a variable may be named at will provided that different qualifiers are added. The qualifiers must be used only within the declaration or specification of the item in question.

3. The scope of non-global entities is the module or block containing their declaration.

4. Except the usual contextual declarations of labels all entities must be declared explicitely.

5. The code of a procedure or operator may be declared with the attribute INLINE.

6. All formal parameters including those of formal procedures must be explicitely specified.

7. In full PEARL storage allocation must be performed dynamically. Nevertheless the compiler may prepare lists of all code and data needed by a task thereby economizing the number of transfer operations between backing and working storage. Cancelling dynamic arrays results in fixed space requirements of a task thus enabling the compiler or the linkage editor to employ storage allocation mechanisms.

8. PEARL provides pointers qualified by the type of the data they point to.
9. All parallel activities constitute tasks.
10. Any statement may be activated as a task. A procedure, however, is not a statement. Thus the unit to be activated in PEARL is not the procedure, but the call of a procedure.
11. The task hierarchy in PEARL can not be determined complet-ely at compile-time.

Solutions of the proposed PL/I subset

1. Module delimiters are not part of the language.
2. Entities to be linked together are declared with the attribute EXTERNAL.
3. The scope of items not declared with the attribute EXTERNAL (possibly by a default-option) is the block containing their declaration.
4. PL/1 provides a large set of implicit and contextual declarations.
5. The language does not contain a means inducing the generat-ion of inline code.
6. PL/1 does not enforce sufficient specification of entry points.
7. The language requires a very dynamic storage allocation mechanism.
8. PL/1 does not provide different pointers for the particular data types.
9. There are different classes of parallel activities.
10. Only procedure calls may be activared as tasks.
11. The task hierarchy of PL/1 is completely dynamic.

Solutions of RTL/2

1. Module delimiters are not part of RTL/2, but belong to a separate control language.
2. Procedures, tasks and data to be accessible to other modules are preceded by the key-word ENT. The modules accessing those entities must contain an appropriate description preceded by the key-word EXT.
3. The scope of non-global entities is the module or block containing their declaration.
4. Except the usual label declarations all items must be explicitely declared.
5. Inline code may be produced by a macro facility included in the language.
6. All formal parameters of procedures must be explicitely specified; this also holds for formal parameters of procedures which are formal parameters themselves.
7. Since the total length of any module is known at compile-time, the compiler can support storage allocation feasible by the linkage editor.
8. RTL/2 includes pointers qualified by the type of the entity they point to (references).
9. 10. 11. RTL/2 does not contain means to generate parallel activities; those are left to the operating system.

Analysis

1. Since module delimiters are needed anyway it is prefersible to include them into the LTPL itself.
2. All languages considered here provide means to indicate, which entities of separately compiled modules are to be linked together. In PL/1, global items may be sparsed over the whole program and be declared at any block level, in LAI, PEARL, RTL/2 declarations of global intities have to be collected and must be put on module level. The use of

additional qualifiers for linking purposes as in PEARL has the advantage of allowing the programmer of a particular module a free choice of identifiers.

3. The conventional scope rules governed by the clock structure of a program seem to be sufficiently lucid and flexible with regard to local items; global items are known within the total module complex which may be considered as a virtual outermost block. It seems not necessary to permit the attribute EXTERNAL within declarations contained in inner blocks thereby abandoning the block-oriented scope structure.
4. All user defined items should be explicitely declared, at least contextually (e.g. labels). The auxiliary implicit declarations (governed by the first letter of the identifier) of PL/1 should be cancelled. It seems dangerous to supplement to much attributes by default.
5. With respect to operating systems involving an automatic paging mechanism the programmer should be able to specify a subroutine with the attribute INLINE in order to provide loading it together with the code where it is called.
6. In order to enable the compiler to check completely, whether formal and actual parameters of a procedure are compatible, LTPL should force the programmer to specify exactly all formal parameters including those of procedures which are formal parameters themselves.
7. Since the actual load of the working storage in an multitasking system is changing dynamically itself a varying length of code and data of a particular task results in a two degree of freedom problem of storage allocation. To avoid this LAI and RTL/2 include no elements of which the space requirements can not be determined at compile-time (e.g. arrays with dynamic bounds to be avaluated on entry in a block). The additional restrictionswith regard to block structure, however, yield no additional gain:

If no dynamic arrays are involved, storage allocation can be performed almost entirely at compile-time. On the contrary, appropriate use of blocks enables the compiler to overlay the variables of parallel blocks. Thus a storage class AUTOMATIC is more economic with regard to storage requirements than a class STATIC and yet not less efficient with regard to run-time requirements provided that all array bounds are evaluable at compile-time (and no recursivity is allowed).

8. Pointers are indispensable for list processing purposes as required e.g. by the preparation of data to be displayed on a CRT. Pointers specific for particular data types yield the possibility of checking the validity of most pointer operations at compile-time which otherwise may cause a lot of overhead at run-time. Furthermore, automatic dereferencing relieves the programmer of writing down explicit operators.

9. Distinguishing interrupt-responses from "normal" tasks results in very machine-dependent programs. It seems preferable to leave the registration or storage of interrupts to the operating system and to start normal tasks on their occurrence. At least the application language should not contain interrupt-tasks, but retain the possibility of user-defined interrupt-responses. In any case, an interrupt-response logically constitutes a separate task which on account of its unlimited core and time requirements can not be performed as part of the OS. Simultaneous I/O-operations are separate tasks as well. Whether the language should provide a distinct form of activation for them is somewhat a question of taste. Since I/O-operations may involve a lot of arithmetic it seems somewhat artificial not to use the common tasking mechanism.

10. Providing activation only for procedure calls is an unneccessary restriction on the programmer. If a task itself does not have parameters, i.e. those stacked for it when the statement causing its activations is encountered, any statement may be executed as a task. Stacking the parameters of a future task, however, is advantageous only in the

case of value parameters where it may be extremely time-consuming. By the use of addresses as task parameters almost nothing is gained in comparison with accessing the addresses when the task is performed.

11. Having a task hierarchy is the more general case, of course. At least asynchronous I/O-operations will logically constitute subtasks in almost every implementation. It is very useful, especially with respect to static storage allocations, to determine the task structure of a module at compile-time.

MAR 1972

Konrad F. Kreuter
Siemens AG Karlsruhe

1972 March

LTPL European Group
LTPLE / o48

Category T

POSITION PAPER ON
EXTENSIBILITY: THE OPERATOR STATEMENT

Summary

In addition to the existing PL/1 features the proposal is being made that operators should be connected with a routine by elaborating a special operator declaration. Operators are identified by the compiler on behalf of the types of their operands. The proposal for the syntactical form of an operator declaration consists out of an OPERATOR-statement followed by a sentencelist, with great similarities to the PL/1 PROCEDURE-statement.

The proposal itself is written beyond the headline 'The OPERATOR Statement' on page 8, where a possible entry-point is to start reading this paper. The explanation of all background to be considered, in a way which consequently leads to the proposal made, is given within the first pages.

It is not very difficult to supplement PL/1 by a programable operator declaration similar to a normal procedure declaration. To devellop and to explain the proposal with a reasonable amount of paper not the normaly used BNF-syntax notation is used but the twofold syntax as introduced to describe ALGOL 68 by A. van Wijngaarden.

Within the Technical Report TR 25.096 edited by IBM Vienna Laboratory dealing with the Concrete Syntax of PL/1 G. Urschler gives the following production rules titeld there with 3.1.3. EXPRESSIONS :

(133) expression ::=
    expression-six | expression | expression-six

(134) expression-six ::=
    expression-five | expression-six & expression-five

(135) expression-five ::=
    expression-four | expression-five
    comparison-operator expression-four

(136) comparison-operator ::=
    > | >= | = | < | < = | ¬> | ¬= | ¬<

(137) expression-four ::=
    expression-three | expression-four || expression-three

(138) expression-three ::=
    expression-two | expression-three
    { + | - } expression-two

(139) expression-two ::=
    expression-one | expression-two {* | /} expression-one

PRECEDING PAGE BLANK-NOT FILMED

(140) expression-one ::=
primitive-expression | { + | - | ¬ } expression-one |
primitive-expression ** expression-one

(141) primitive-expression ::=
(expression) | reference | constant | isub

These production rules define the monadic and dyadic operators to be used in the language PL/1. Moreover the precedence of the different operators is fixed. But nothing is stated out about the data-types to be connected together using any particular operator and what kind of result this operation delivers. The way ALGOL 68 is defined within the Report on the Algorithmic Language ALGOL 68 by A. van Wijngaarden et al. (Numerische Mathematik, 14, 79-218 (1969)) enables the lay down of every possible type of operands as well as the type of result using only syntactic and programming language features but no semantic addition. To achive the same result with the PL/1 production rules listed above they must be rewritten.

First thing to be done is the extraction of all operator symbols the way production rules (135) and (136) are an example of:

(3.7) expression-seven ::=
expression-six | expression-seven
precedence-seven-operator expression-six

(3.6) expression-six ::=
expression-five | expression-six
precedence-six-operator expression-five

(3.5) expression-five ::=
expression-four | expression-five
precedence-five-operator expression-four

(3.4) expression-four ::=
expression-three | expression-four
precedence-four-operator expression-three

(3.3) expression-three ::=
expression-two | expression-three
precedence-three-operator expression-two

(3.2) expression-two ::=
expression-one | expression-two
precedence-two-operator expression-one

(4.8) expression-one ::=
primitive-expression | monadic-operator expression-one |
primitive-expression precedence-one-operator
expression-one

In addition the production rules for the operators itself:

(4.7) precedence-seven-operator ::= |

(4.6) precedence-six-operator ::= &

(4.5) precedence-five-operator ::= >| >=| =| <| <=| ¬>| ¬=| ¬<

(4.4) precedence-four-operator ::= ||

(4.3) precedence-three-operator ::= + | -

(4.2) precedence-two-operator ::= * | /

(4.1) precedence-one-operator ::= **

(4.0) monadic-operator ::= + | - | ¬

The monotony of all the production rules from that one producing expression-seven down to that one for expression-two suggests to combine all six rules together leaving one and only one rule. The model to do so again is the representation of ALGOL 68 with its twofold grammar. The de-

tailed explanation of the following production rules is to be taken from chapter 1 of the ALGOL 68 report accordingly.

Rule (3.7) to (3.2) are transponed to:

(4.9) expression-PRECEDENCE ::=
expression-PRECEDENCE-minus-one | expression-PRECEDENCE
precedence-PRECEDENCE-operator
expression-PRECEDENCE-minus-one

with the additional second-level production rules:

(5.o) PRECEDENCE ::=
TWO | THREE | FOUR | FIVE | SIX | seven

(5.2) TWO ::= THREE-minus-one ::= two

(5.3) THREE ::= FOUR-minus-one ::= three

(5.4) FOUR ::= FIVE-minus-one ::= four

(5.5) FIVE ::= SIX-minus-one ::= five

(5.6) SIX ::= seven-minus-one ::= six

(5.1) two-minus-one ::= one

The production rules (4.o) to (4.9) together with the rules (5.o) to (5.6) define exactly the same as the original rules (133) to (14o) do. First above this means that until now some semantic statements are to be made about the action a particular operator shall take upon specific operands. To remove this unconvenience it must be thought of and stated out that an operator is connected with a routine which might be in case of a dyadic operator the body of a function requirering two parameters. The term 'precedence-PRECEDENCE-operator' from rule (4.9) must be defined by:

(5.7) precedence-PRECEDENCE-operator ::=
dyadic-operator ::=
procedure-with-Lparameter-and-Rparameter-precedence-PRECEDENCE-operator

Using the terms 'Lparameter' and 'Rparameter' only the fact should be noted that operands being the parameters of a routine are written leftside and rightside of the operator symbol in question. Both terms finally become simple parameters.

In a similar way rule (4.o) requires an addition:

(6.o) monadic-operator ::=
procedure-with-Rparameter-monadic-operator

The connection between an operator symbol and the correlated routine is achieved by an operator-declaration which is to be regarded as some kind of a procedure-declaration. This does not imply that an operator-declaration must be available for the normal user as a language feature.

It is state of the art to say definitely what types of parameters a procedure should be able to receive as well as what type of result should be delivered if any. As a consequence the identifier of a particular procedure is connected inseperably with the types of parameters once choosen. This means that to a second procedure a new identifier must be given if parameters of different types are to be passed over although the body of that procedure might be the same as that of the first one. Looking at the definition of an operator one could expect a necessity to use one plus-symbol notation to add integer numbers and a different one to add real numbers. Such a direction today would be very unconvenient and impractible. In the contrary one and the same operator symbol

must lead to different routines depending on the types of operands in question. An importand feature of PL/1 is the GENERIC attribute which allows the programmer to use one and the same identifier for different procedure bodys but defined and selected by means of the different types of parameters.

To take into account the dependence of an operator routine upon the types of its operands a special syntax notation can be choosen. A model is ALGOL 68 again. The production rule (4.9) for expressions gives after integratingthe types of operands

(7.o) MOID-expression-PRECEDENCE ::=
MODE-expression-PRECEDENCE-minus-one |
LMODE-expression-PRECEDENCE dyadic-operator
RMODE-expression-PRECEDENCE-minus-one

The same extension to (5.7) and (6.o) results in

(7.1) dyadic-operator ::=
procedure-with-LMODE-parameter-and-RMODE-parameter-
MOID-precedence-PRECEDENCE-operator

(7.2) monadic-operator ::=
procedure-with-RMODE-parameter-MOID-monadic-operator

and to complete the whole arrangement rule (4.8) as

(7.3) MOID-expression-one ::=
MODE-primitive-expression |
monadic-operator RMODE-expression-one |
LMODE-primitive-expression
procedure-with-LMODE-parameter-and-RMODE-parameter-
MOID-precedence-one-operator
RMODE-expression-one

Together with rules (5.o) to (5.6) and with regard to RMODE ::= MODE and LMODE ::= MODE the production rules (7.o) to (7.5) supersede the initially listed rules (133) to (14o) taken out of TR 25.o96 if some additional arrangements are made about the connection of operator symbols and the related routines. (see page 11 for MODE)

To illustrate the production rules develloped above using a twofold grammar the following expression shall be parsed:

a + 1o

where a is declared as a real variable. Now the symbol + must lead to a routine with one real and one integer parameter delivering as result a real number, that is a

procedure-with-real-parameter-and-integer-parameter-real-precedence-three-operator

The expression in question therefore is a

real-expression-three ::=
reference-to-real procedure-with-real-parameter-and-integer-parameter-real-precedence-three-operator constant-integer

Note that the initially used rule (141) has changed as mentioned later on.

The OPERATOR Statement

There are operators requiring only one right operand - called monadic operators, and there are operators requiring one left operand and one right operand together - called dyadic operators.

In addition to the existing PL/1 features we propose that operators are connected with a routine by elaborating a special operator declaration. Operators are identified by the compiler on behalf of the types of their operands.

Introducing the operator declaration the appearance of an operator symbol within an OPERATOR-statement as part of an operator declaration is provided. In opposite to normal declarations of variables for example which may appear only once within the range defining on particular identifier, more than one operator declaration must be permitted in such a range to contain the same operator symbol.

Because there is the possibility to get access to different entry points of one procedure with one distinct procedure-identifier nowadays - using the GENERIC attribute, and by this way to select the special entry points while regarding the specific types of arguments - we propose that the appearance of more than one operator declaration containing one and the same operator symbol, is to be regarded as a combination of a number of declarations similar to such procedure declarations and one declaration with a GENERIC attribute belonging to that particular operator symbol.

The authors proposal for the syntactical form of thuch an operator declaration, consisting out of an OPERATOR-statement followed by a sentencelist, is as follows

```
(9.o)  operator ::=
          OPERATOR { operator-indicant | operator-token }
          ( [ identifier [dimension-attribute ][attribute...]
          , ] identifier [dimension-attribute ][attribute...])
          RETURNS ( [ dimension-attribute ] attribute... )
          [precedence-declaration ] ; sentencelist
```

with similarities to rule (2) out of TR 25.096 dealing with procedures. In addition to this the following rules are necessary

(10.0) precedence-declaration ::=
PRECEDENCE = { digit | operator-token }

(10.1) operator-indicant ::=
. identifier .

(10.2) operator-token ::=
⊥ | & | > | >= | = | < | <= | ¬> | ¬= | ¬< | || | + | - | * | / | ** | ¬

For all the operators provided in the PL/1 language to be used within a program there are operator declarations allocated in the surrounding system - from programmers viewpoint - as procedure declarations for e.g. trigonometric functions can be regarded as. If there are only such declarations nothing has changed with PL/1. But if thuch a declaration is permitted to be written within a particular program -

that is the proposal to be made here

- the language is what we call extensible. Looking onto its behavior the operator declaration is the same as a declaration of a function subroutine with two parameters.

The PRECEDENCE attribute as part of the OPERATOR statement is optional. This attribute must appear only once if ever, though more than one declaration using the same symbol is programmed. Monadic operators are connected with a PRECEDENCE = 0 attribute. A possible supplement to (5.0) is to allow PRECEDENCE = 8 and 9. An operator token instead of a digit states the same precedence to be given to the new operator as that one represented by the token.

Together with production rule (7.3) on page 8 the following two rules should be mentioned

(7.4) <u>MOID</u> ::= <u>MODE</u> | void

(7.5) <u>MODE</u> ::=
every possible type attribute

As an example of an operator declaration we give the redeclaration of an adding operator to provide integer addition:

```
OPERATOR .PLUS. ( I BINARY FIXED, K BINARY FIXED )
   RETURNS ( BINARY FIXED ) PRECEDENCE = 8;
   I = I + K; RETURN (I); END;
```

Now, the following to statements are interchangeable

```
J = M + 13 ;

J = M .PLUS. 13 ;
```

UNIVERSITE PARIS - SUD Aug. 26th 1972

OPERATING SYSTEMS INTERFACE

G. Verroust

1/ Introduction

Of course, this paper is not exhaustive. It may be considered as an introduction to a discussion on the problem. It is a set of remarks and proposals based on my own experience and problems, and some discussions with members of LTPL-E group. System interface is, with I/O handling the most important problem in Real-time languages.

In this paper, I refer to PL/I for the following reasons :

-- PL/I is now the first universal procedural language that is really operational on several modern computers.

-- There is no contradiction between our aims and PL/I features, but only deficiencies of PL/I.

It seems reasonable, rather than create a completely new language, to take the same conventions as PL/I to describe the same functions and to extend (and modify if absolutely necessary) it.

Another inconvenient of PL/I, due to its great number of possibilities is its size, the size of its compilers. As there are, in process control, many users of minicomputers, there is a need for an universal subset of PL/I. However, many people now have access to a big computer and it is very convenient and inexpensive to compile on a big machine a program for a minicomputer (Higher level language usable, better optimization of generated code etc..).

Some PL/1 subsets exist(CPL/I,..)but a normalization should be desirable.

PL/I is not a real-time language but a batch processing language for modern systems (File handling, I/O channels etc ...). The purpose of a programming language is to describe an algorithm. This algorithm may be applied on data entirely defined in the program or on data written on devices specially designed for data handling. In real-time processing, in process control, the algorithm must act on various material devices, at a correct time and handle data coming from all kinds of gimmicks. That's the LTPL problem.

2/ General principles

The problem of the linkage between LTPL and an operating system is situated at several levels.

-- The description of the ways used by the operating system to realize physically the logical operations described in the language.

-- The use, in the language, of all the physical features of a particular system.

-- The ability to use the language as a system language to write or generate, for instance, a monitor.

The purpose of an Operating system is to automatize the management of a computer. It is a set of programs and logical methods. An operating system manages the jobs and tasks, the libraries, links the physical I/O devices and the logical files, manages the resources , optimize the use of the computer.

In a real-time system, it is necessary to access, from the programming language to all the physical features of the computer and its peripheral units.

3/ System features in the language

The programmer must be allowed to specify to the compiler optimization criteria for computation.

The channel handling and multitasking facilities of PL/I may be easily extended to permit the implementation of real-time tasks.

All the ways of interconditioning tasks and I/O may be described in PL/I by the very versatile handling of the EVENT parameter. It must be said that

many of the facilities of the language are not yet implemented but may be used in process control systems (Boolean relations between events for instance) and will be used on the future multiprocessors. However, there is a need to associate a task with an external event declared by the programmer. This is possible only by initiating a fictive I/O and handling the event associated with the end of I/O.

As we said in our 11th meeting, there is a need for new logical classes of I/O used in process control. These classes will be defined after a serious theoretical analysis.(Analog inputs,ADCs,thresholds...)

It should be very useful to give control to real-time tasks from user defined ON conditions. In PL/I, ON conditions are used now only to handle abnormal occurences during the execution of a procedure.

There a need for a complete set of timer handling instructions usable in conjunction with event activation of tasks.(Periodical activation of a procedure, limitation of the duration of a task...)

A mean to mask and modify the priority of interrupts should be very desirable. This is not implementable on IBM360, but may be useful on real-time machines.

With simultaneous access of data by several tasks, there is a need for the definition of semaphores associated with memory reservations. This semaphore may be a key allowing access to this memory only to tasks with the same key. The semaphore may be a generalization of a PL/I EVENT. It should be very powerful because it should permit the automatic reactivation of a task at the liberation of the protected memory.

LTPL must give the control of the memory residence of a task or a procedure. This exists in PL/I but becomes now very important. Some machines (Burroughs,IBM) having automatic virtual memory handling, it must be possible on these machines to impose a permanent ferrite core (or MOS) residence to a part of the program to reduce the real-time response to random events.

The access to the swapping or overlaying facilities of the operating system should be appreciated in some cases.

The LTPL must be able to describe algorithms performed on 4th generation structures of computers that are already implemented on some real-time systems.(Multiprocessing, specialized satellites, recorded firmware...)

For instance, it is necessary to extend the attributes of a task or a procedure to specify its execution on a satellite computer(With the loading of this computer from the central machine).

There is a problem with specialized data processing devices used as peripheral units(for instance pulse height analyzers in nuclear physics).

Such a unit may be described in the program in two simultaneous ways :

-- A non-executable procedure describing the hard-wired program of this peripheral processor.

-- A small executable procedure containing the I/O linkage with this unit.

The same method may be used for units as array processors or hard-wired sort/merge units.

To conclude this point, in the attributes of a procedure, it must be possible to give a complete information on the mode of execution of this procedure. (Environment, computer used, I/O associated ...)

It will be desirable to use LTPL to write microprograms for recorded firmware computers. This is a big problem that will be examined later. For instance, a compiler may define the set of instructions well adapted to execute an algorithm, define the correspondings microprograms, create and load them.

Graphical devices handling is a special very important problem and some extensions of PL/I are being done.(Or of other Algol-like languages, the procedural structure being well adapted for picture handling).

4/ LTPL and Nucleus

The use of LTPL as system generation language should be very useful. It should allow the correspondance between the physical devices of the computer and their handling instructions in LTPL. I would insist on the interest of generating with the nucleus a descriptive block of the system features, this block being used by the

compiler to verify the feasibility of the program on this system and giving warning messages on the program listing. There thus a need for a normalized description of all the possible hardware features. For instance, if the computer has associative memories, the compiler must be able to use them with adapted instructions, but if this program is run on a computer without associative memories, the program will be executed with a simulation by the generated code.
I would insist on the very big interest of this normalized hardware-descriptive language for the implementation of transportable real-time languages compilers. This may give a solution to the problems set by the inherent non-transportability of those "ENVIRONMENT" attributes in PL/I.

In conclusion, it is needed a subset, a macro-language or an extension of LTPL to perform the nucleus generation on a particular system.

We always must remember that in real-time an algorithm is strictly configuration and machine dependent. Generally, a real-time program is essentially untransportable.

5/ LTPL Pre-processor

One may understand that a pre-processor is a very powerful feature for LTPL.

PL/I has a pre-processor allowing the means to extend or modify keywords to implement, for instance, a specialized professional programming language (For architects, biologists...). One of the greatest interests of the PL/I pre processor is that it uses a subset of the PL/I. But this subset is too restricted.

A great attention must be given to the possibilities offered by the access to the files and librairies of the system by the preprocessor.

Such a feature may give the language a very powerful extensibility. For instance it becomes possible to generate, from new keywords, an assembly subroutine called from the program and automatically assembled and linked with the program.

The normalization of macros and subroutines used in different systems for similar functions is a problem

concerning professional users groups of the language.

However, a great attention must be given to the danger of predictability loss by bad use of the macro processor. A new important feature, universally used, must lead to an extension of LTPL and must not be implemented always in the pre-processor.

For instance, it is not possible to implement a real-time language with the conventional PL/I and its pre-processor because there is not a conventional use of a computer, but a way very different than batch processing for which PL/I was designed.

Process control is not a new class of applications but a new way of using a computer.

8/26/72

SECTION VI

REVIEWS OF LTPL DEVELOPMENT WORK

As with the work on FORTRAN and on problem oriented languages, the Japanese members have been very helpful to their colleagues from other countries by reviewing the work of the others. The Japanese have also made considerable contribution to the work of the several Workshop committees in their own right as well. Two examples of the review work concerning the LTPL are presented here. Both are from the Long Term Procedural Language Committee-Japan chaired by Dr. M. Sudo. Both are from the *Minutes of the Purdue Workshop on Standardization of Industrial Computer Languages*. The first from the *Fourth Minutes* (pp. 208-213), the second from the *Eighth* Minutes (Appendix IV-2, pp. 179-188).

COMMENTS TO THE LONG TERM PROCEDURAL LANGUAGE COMMITTEE

Working Group for the Long Term
Procedural Languages
Technical Committee on Industrial
Computer Languages of Japan
Chairman, M. SUDO

Although procedural languages belonging to the scope of investigation by the Long Term Procedural Languages Committee are offered from some of vendors in Japan, they have not yet gained the principal position of the industrial programming languages. Most of users application programs are, at present, written in assembly language and FORTRAN with some extension for real time use. Many people, however, recognize importance and necessity of standardization of procedural language beyond the concern of the Interim Procedural Languages Committee. We consider that the activities along the direction of standardization of industrial computer languages through workshops at Purdue University should be highly appreciated.

Followings are the comments and suggestions to the report of the Long Term Procedural Languages Committee at the 3rd Workshop, concluded through discussions at the Working Group for the Long Term Procedural Languages of the Technical Committee on Industrial Computer Languages, organized at JEIDA (Japan Electronic Industry Development Association), Tokyo.

PRECEDING PAGE BLANK-NOT FILMED

1. Basic Design Assumptions

1.1 Usage of the Languages

We agree to the basic assumption that the language is intended for both the applications and system's programmers usage with the primary emphasis being given to the former. The principal objective of the language which is to be proposed as a standard should be to provide an effective and economical programming tool for the application programmers.

Before examining the individual item of the general specification, we put following assumptions on the role of this language.

(1) Object programs generated from a compiler of the language are executed under the control of an executive (or monitor) program.

(2) Object programs may be connected to some existing program packages.

(3) Whole of an application program should be written in the language with no required use of assembly language.

1.2 Size of the Language

We agree to the assumption that the implementation of a compiler for the full language set should be possible on what is currently considered to be a medium scale configuration of an industrial computer.

We are afraid, however, that the size of the language described in the general specification may be a little too large to implement a practical compiler for a configuration with 16K core memory.

2. On General Specification

2.1 Background of the Language

The language seems to be defined using PL/I as background and to have the structure quite similar to PL/I. It is a fact that PL/I is a promising language and is expected to fill the demands which have not been gained with existing procedural languages. On the other hand, it is pointed out that the versatile features of PL/I would require a lot of resources both on implementation of a compiler and on run-time packages.

We recognize that the general specification of the proposed language is built up through putting restrictions to PL/I full set from the industrial applications viewpoint. It seems necessary to add more restrictions on some features of the language, in order to make the language specification practical for the medium size industrial computers.

Items to be investigated on adding restrictions are ——:

(1) Ability to operate on address variables, and data type declaration of locator variable.

These are useful especially to system writing but will not be necessary for industrial applications.

(2) Named hierarchical data structure.

This feature would be useful to applications programmers. But we consider it is quite difficult to implement this feature efficiently and most of practical compilers offered from vendors for this language would provide only some restricted usage of data structure.

(3) Label variables.

This can be replaced with the connection of some simpler language units.

(4) Implicit multi-precision.

This would require complicate compiling procedures and a lot of run-time packages. For practical use it would be tolerable to write multi-precision designation explicitly.

2.2 Interconnection with Operating System

It is one of features of industrial programming languages to have an explicit interactions with operating system and/or equipments.

Our comments on those parts of the general specification of the language which cover the interactions with operating system are as follows.

(1) Statement should be of as general form as possible.

Operating systems are somewhat machine oriented and a whole set of statements defining the interactions with operating system does not seem possible to be selected uniformly among machines at present, because the functional specifications of operating system are to be far from standardization.
We suggest, therefore, to introduce a concept of classes of executive functions (such as Supervisor Call) as a standard for defining interactions with operating systems, and to leave further detailed specifications to implementors.

(2) Standard form of statements should have some flexibility so that it would cover differences in machine size and system design architecture.

A standard form may be defined with typical number of parameters. But some machines may have less number of items to be specified than the standard, and other may have needs for more items to utilize their resources efficiently.

In order to cover these cases we suggest that the standard of the language should consist of two elements - typical form of statements and the general rules to give changes for individual implementor.

2.3 File Description

Basically we agree to the language to provide the ability to define file attributes referred to in the general specification, and we have a comment, which is similar to that given from engineers in Netherland, that core files should be considered in the file system.

We have a question, however, if file descriptions would be fully standarized within individual procedural language. Standardization of file descriptions as for structure and method of processing should naturally be investigated independently from individual language. This workshop concerning industrial languages may, we think, not be responsible for the common file standardization.

2.4 Program Structure

Besides the storage allocation statements, the language should have some declaration statement concerning reenterability of program segments. We consider this may be optional feature.

2.5 Language and Compiler Interaction

We consider it is a good attempt for the general specification of the language to contain paragraphs concerning "language and compiler interaction" and "debugging and error checking."

The general specification should refer to the language features and/or programming tools useful for on-line compilation and on-line debugging both of which may be often required at site.

In these cases considerations should be paid upon machine configuration and resource utilization.

2.6 Concepts of Level and/or Subset of Standards

Is the standard of the long terms procedural language to be realized as single level? Is it intended to consist of several levels, like FORTRAN or ALGOL?

We expect that smaller scale computers will be widely used for industrial applications in near future, and the standard procedural language will be required to cover the usage of these machines as well. Since the proposed standard language may be, we consider, somewhat higher grade, it will be necessary to provide a lower grade standard for this language. It may be sufficient to define method of selecting a language subset from the full set.

COMMENTS ON ASSESSMENT OF PL/1 AS A BASIS OF LTPL DEVELOPMENT

August 1972

Japanese Technical Committee
on Industrial Computer Languages

Working Group for LTPL
(LTPL-J)

1. Introduction

To investigate the suitability of PL/I as a basis of LTPL development LTPL-J set up eight areas to study mainly according to the items of the functional requirements of LTPL agreed at the sixth workshop meeting. These items were assigned to more than ten members of LTPL-J. Some items were therefore assigned duplicatedly and studied by two members independently. After each position paper was proposed, we discussed every topic one by one. The BASIS/1 document (Version 6) issued from the ECMA-ANSI co-working group was used as a text book of PL/I.

Hereafter is a summarized report of our discussion through LTPL-J meetings at JEIDA, Tokyo.

PRECEDING PAGE BLANK-NOT FILMED

2. General

2-1. System writing features, by which we mean features for describing operating systems and/or hardware, should not be the major aims of the standard LTPL because LTPL compilers and run-time packages should remain within a practical scale for medium size industrial computers.

LTPL features should be defined as much independent as possible from O.S. and hardware descriptions.

2-2. In order to implement practical LTPL compilers much of BASIS/1 features should be removed or modified. The meaning of the word "practical" can be stated as that LTPL compiler with run-time packages is not so much larger than those of the extended FORTRAN.

2-3. It seems necessary to have a few levels of the language for practical applications for various scales of machines including mini-computers.

3. Data Types and Manipulations

3-1. Binary and Decimal

Data type attribute DECIMAL should be removed. Thus, only BINARY is remained, and the selection of BINARY/DECIMAL becomes meaningless. The internal representation of data is restricted to Binary.

3-2. Manipulating bit data

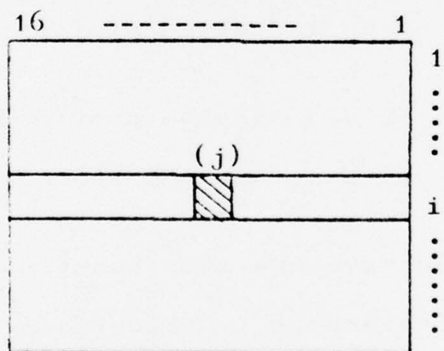
Bit data are manipulated in BASIS/1 always as bit strings. Though a single bit can be represented as a bit string of length 1, it is troublesome to access single bit always through built-in function SUBSTR.

For LTPL applications it is more suitable to represent binary data as data structures composed from binary digit.

For example the hatched bit in the figure below should be accessed directly by a notation

B (i, j)

instead of using SUBSTR function like BASIS/1.



3-3. Checking facility for bit status

It seems necessary to provide a function SCAN (B) or an equivalent statement to check bit status, where B denotes a bit string and by executing SCAN (B) the contents of B are checked bit by bit from the left giving the count of bit number of the first non-zero.

3-4. To reduce the complexity of implementation some restrictions should be applied to the followings:

a) types of storage

b) picture data

c) mixed operation.

3-5. Hexadecimal/Octal notations should be introduced as a representation of literals consisting of bit strings.

4. Static Program Structure

4-1. It is necessary to provide a method of detection of the external procedure which is called at first.

4-2. Call by value should also be provided for transferring parameters in calling functions and subroutines.

4-3. DEFINED should have the same function as EQUIVALENCE in FORTRAN. By this feature it is allowed to allocate data with different attributes to the same storage.

4-4. BASIS/1 seems to have an assumption of the existence of a single program under the control of the operating system, though more than two tasks can exist in the program.
In LTPL applications a number of programs must run (as tasks) under the operating system.

4-5. Data with EXTERNAL attribute should be devided in two groups: common data among the external procedures within a program, and common data among the programs.

It was proposed in LTPL-J to devide EXTERNAL attribute at the level of LTPL into EXTERNAL and GLOBAL correspohding to the two. But this was not resolved because there is another proposal to provide these as the functions of operating system.

5. Dynamic Program Structure

5-1. Independence of tasks

Following are pointed out associated with item 4-5.

a) Parameters should be transferred by value.

b) Extermination of a parent task should have no action upon the associated tasks.

5-2. RECURSIVE option is not necessary.

5-3. AUTOMATIC attribute is not necessary in the LTPL applications because this attribute implies an amount of storage requirement in the run-time and this may possibly reduce the merit of the attribute. Moreover some increase of overhead time is expected.

6. Resource Management

6-1. LOCK/UNLOCK statements

It is desirable to extend the scope of LOCK/UNLOCK statements to other resources as much as possible. This will be a means of preventing system dead locks. These statements should be able to have more than two arguments.

6-2. A means is necessary to check whether a resource is occupied or not.

7. Tasking

7-1. As for tasking we have an agreement of the functional requirements of the 6th workshop. Tasking facilities of BASIS/1 should be replaced with a new one based on our agreements.

7-2. On the report of LTPL tasking agreed at the 6th workshop, following change are proposed by some LTPL-J members.

a) Selection of the basic status of a task system has a dependency on the structure or phylosophy of the operating systems. In some systems IDLE should be the basis and in another systems DORMANT is more suitable. Thus it should be allowed to regard the arrows coming to DORMANT as coming to IDLE, and to add some words stating on the freedom of state selection between IDLE and DORMANT.

b) It should be clarified that the dependency of a task upon its ancestor in BASIS/1 is removed.

8. Input/Output

8-1. To facilitate a various types of devices and modes of input-output, a generalized syntax is required so that the parameters specify the wide variety of input output operations. I/O facilities should be provided for character I/O, process I/O, graphic I/O, and communication I/O.

8-2. Computer linkage

I/O of LTPL should have no dependence upon the mode of communication, i.e., processes depending on communication modes should be left to implementors.

a) Output

Output data should be available both from the core and from the file. Data from the core may be specified by the variable in FROM option designating the location of the data. Data from the file may be specified by the variable in KEY option designating the location of the key. Parameters such as data station names and terminal names should be included in data and processed according to the rules of communication. LTPL program should have no specification about them.

b) Input

Input data should be put on a specified file and the input task is called with the key of data as its parameter.

Input statement might have INTO option causing a transfer of data to the designated area. The former serves to the contention processing, and the latter to the polling processing.

8-3. Locking of files should be studied.

8-4. Syntax of I/O statements should be so flexible as to cover the appearance of some new type of I/O.

9. Interrupt Handling and Synchronization

9-1. BASIS/1 does not seem to be designed putting much of weight on processing of external interrupts. Means should be studied for connection and disconnection between external interrupts and tasks or events.

9-2. As for the conditional interrupts there are some cases in which the occurrence of interrupts such as FIXED OVERFLOW, OVERFLOW, UNDERFLOW, and ZERODEVIDE are natural in a circumstance. To process these cases properly it may be required to exchange the default rules of condition prefix, enable/disable.

10. Default

10-1. The default rule of storage class should be changed to STATIC.

11. Concluding Comments

Japanese group observes that PL/I is able to be a basis of LTPL development if some modifications are properly made from the point of view of industrial applications.

Modifications to PL/I may be required in many features of the language, but major ones are concentrated to the functions for real-time processing and multi-task processing.

Another pcint to note in the development of LTPL is the scale of the language and of the translators. If LTPL is expected to be used in medium or small scale industrial computer applications, it is necessary to cut an amount of the language features off. The ANSI subset of BASIS/1 seems to be too large for such LTPL applications.

It may be necessary to have two or more levels of LTPL standards to cover the various scale of applications.