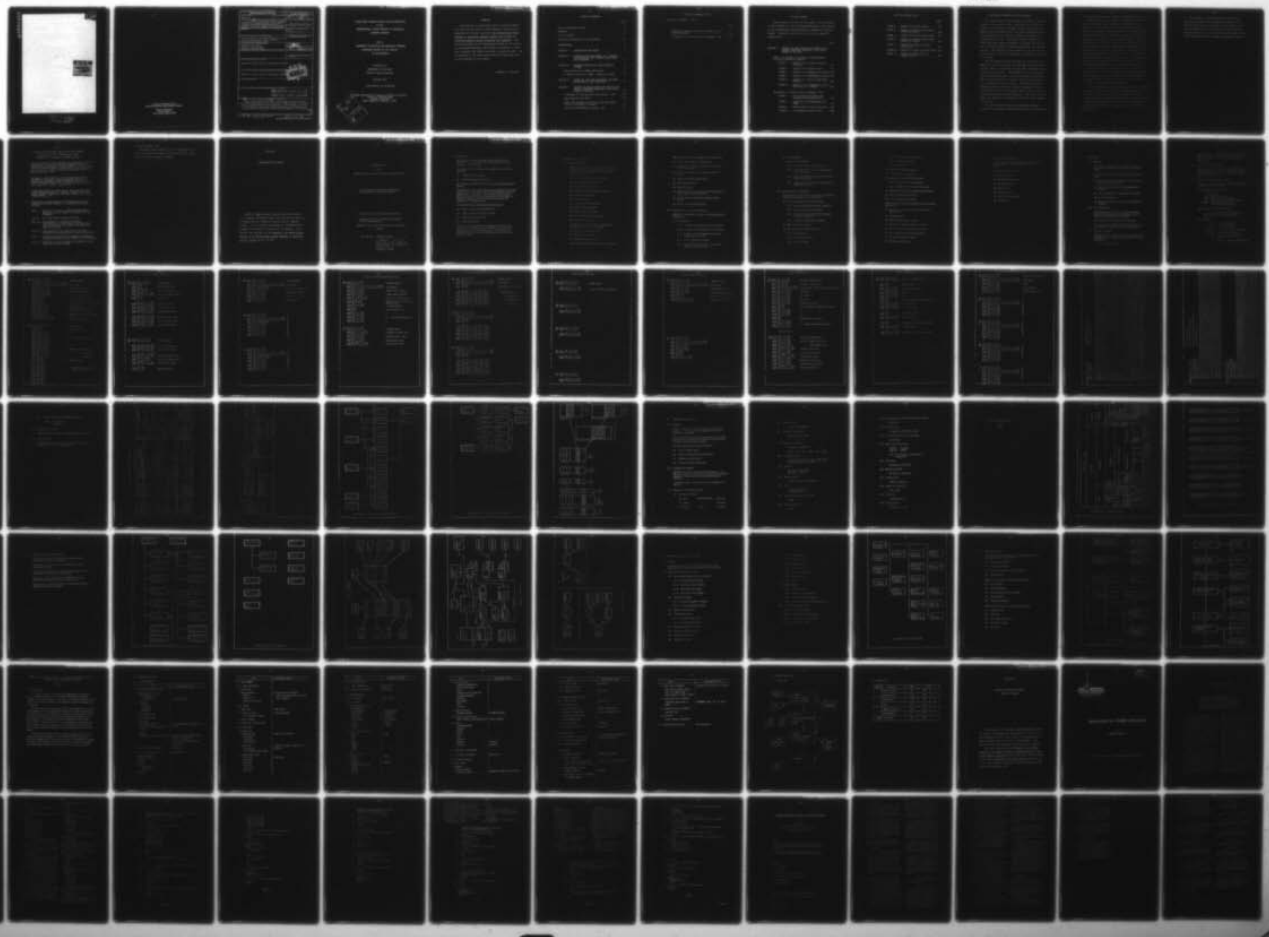
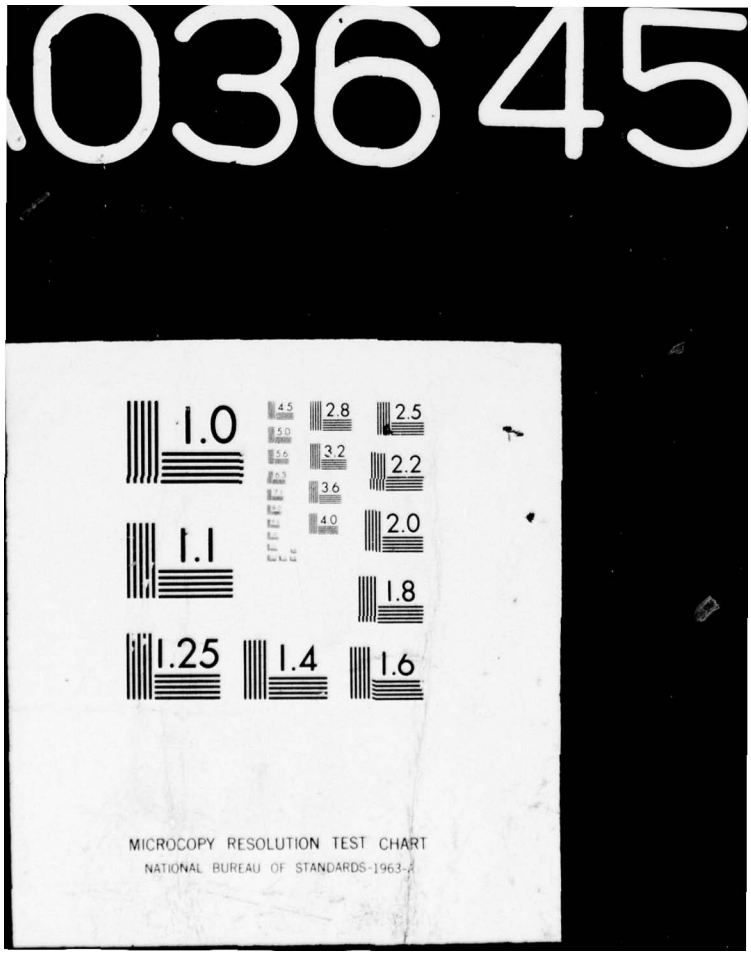
AD-A036 455 PURDUE UNIV LAFAYETTE IND PURDUE LAB FOR APPLIED IND--ETC F/G 9/2
SIGNIFICANT ACCOMPLISHMENTS AND DOCUMENTATION OF THE INTERNATIO--ETC(U)
JAN 77 N00014-76-C-0732

UNCLASSIFIED NL

1 OF 3
AD
A036455




036 45
1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A036455

(12)

DDC
MAR 7 1977

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

(12)

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: NR049-388	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER: (9) Final rept. 1 Feb 76 - 31 Jan 77	
4. TITLE *(and Subtitle)*: (6) SIGNIFICANT ACCOMPLISHMENTS AND DOCUMENTATION OF THE INTERNATIONAL PURDUE WORKSHOP ON INDUSTRIAL COMPUTER SYSTEMS, - PART V. (9) DOCUMENT ON EXISTING AND PRESENTLY PROPOSED LANGUAGES RELATED TO THE STUDIES OF THE WORKSHOP.	5. TYPE OF REPORT & PERIOD COVERED: Final 2/1/76 - 1/31/77
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s)	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	12. REPORT DATE: (11) Jan[illegible]977
	13. NUMBER OF PAGES: 284 + xiv (12) 265 p.
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*	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)*

COPY AVAILABLE TO DDC DOES NOT PERMIT FULLY LEGIBLE PRODUCTION

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

This volume represents Part V 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 1 JAN 73 1473 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 V

DOCUMENTS ON EXISTING AND PRESENTLY PROPOSED LANGUAGES RELATED TO THE STUDIES OF THE WORKSHOP

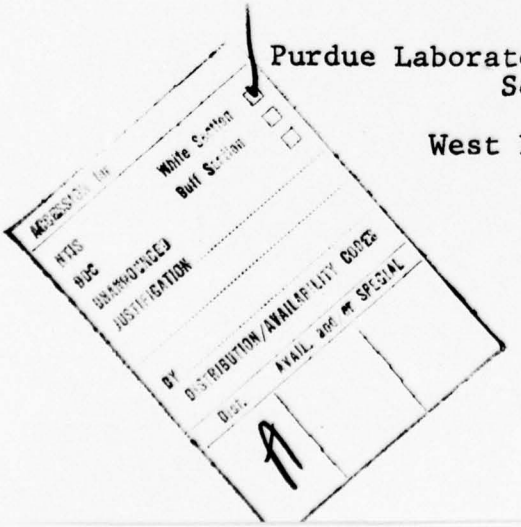
Prepared for

Department of the Navy

Office of Naval Research

January 1977

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

Background Information on the Workshop. xii

INTRODUCTION. 1

SECTION I - INTRODUCTION FOR PROCOS 5

SECTION II - OUTLINE OF AN EXPERIMENTAL PL/I COMPILER IMPLEMENTATION FOR A MEDIUM SCALE INDUSTRIAL COMPUTER. 63

SECTION III - STANDARD SOFTWARE FOR CAMAC-COMPUTER SYSTEMS 75

Specifications for CAMAC Subroutines 77

Standard Software for CAMAC - Computer Systems . . . 89

SECTION IV - REPORT ON REAL-TIME EXTENSIONS AND IMPLEMENTATIONS OF REAL-TIME BASIC 95

SECTION V - REPORTS ON SOME EXISTING AND PROPOSED PROCEDURAL LANGUAGES IMPORTANT IN THE DEVELOPMENT OF THE LTPL. 179

A Language for Writing Real Time Systems - LTR . . . 181

Some Proposals for RTL/2 189

PEARL, the Concept of a Process- and Experiment-oriented Programming Language. 217

Industrial Programming Language - LAI. 231

TABLE OF CONTENTS (Cont.)

SECTION V - REPORTS (Cont.)

Page

Elementary Description of the Features of the PROCOL Language 243

Development of Process Control Language - PCL . . . 305

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. Therefore they will not be in strict numerical sequence here.

Page

SECTION V - REPORTS ON SOME EXISTING AND PROPOSED PROCEDURAL LANGUAGES IMPORTANT IN THE DEVELOPMENT OF THE LTPL

PEARL, the Concept of a Process- and Experiment-oriented Programming Language

FIGURE 1 - Example of a Circular Concatenated List 221

FIGURE 2 - Scheme of an "Address Tree" . . . 222

FIGURE 3 - Example of a RANDOM Point Display 228

FIGURE 4 - Example of an INCREMENTal Display 228

FIGURE 5 - Example of a Vector (LINE) Mode Display 228

FIGURE 6 - Example of an INTERPOLated Curve and an Arc of a CIRCLE. 228

Development of Process Control Language - PCL

FIGURE 1 - Hitachi Control Computer 500 Language Processing System. . . . 278

FIGURE 2 - Usage Rate of Programming Language 278

FIGURE 3 - Memory Layout of Data Structure . 279

FIGURE 4 - Bit Dimension and Bit Data. . . . 280

LIST OF FIGURES (Cont.)

Page

FIGURE 5 - Example of Decision Table 281

FIGURE 6 - Example of Coding for Decision Table Processing. 281

FIGURE 7 - Concept of Virtual Memory Space . 281

FIGURE 8 - Example of Coding Using Virtual Memory Data Space 282

FIGURE 9 - Example of Coding for Color CRT Display 283

FIGURE 10 - Example of Usage of Compile Mode. 283

FIGURE 11 - Example of Compilation by Compile Mode %. 284

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 cosponsored 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 American 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.

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. The 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 severally 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.

In their review of the several types of industrial computer languages under consideration by the several Workshop language committees the attached reports on presently proposed or recently developed languages were presented to the Workshop and published in the several Minutes. They are reproduced here to document the state-of-the-art against which the LTPL, the POL translation systems, and the FORTRAN

and BASIC extensions are being developed. The language comparison document of LTPL-E (Item 3 of Table I) is also important here.

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 Systems 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 Industrial 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.

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.

SECTION I

INTRODUCTION FOR PROCOS

PROCOS or PROcess Control Oriented Software System is a language developed in Japan over the past few years as a standard POL for industrial process control computer systems. It is a "fill-in-the-blanks" or "format-defined" language (see Section V of Part I of the Summary). This document was published in the Minutes of the Second Annual Meeting of the International Purdue Workshop on Industrial Computer Systems on pp. 225-276.

PRECEDING PAGE BLANK-NOT FILMED

INTRODUCTION

FOR

PROCOS (Process Control Oriented Software System)

(If there are any questions or comments on these documents, please let us know.)

Problem Oriented Language Working Group

Technical Committee on Industrial Computer Software Standardization

Japan Electronic Industry Development Association

(JEIDA)

WG Chairman; Takashi Tohyama

Instrumentation and Control Dept.
Chiyoda Chem. Eng. & Const. Co.,
No. 1580, Tsurumi
Yokohama, JAPAN

PRECEDING PAGE BLANK-NOT FILMED

1. Introduction

The Problem oriented Language (POL) Working Group developed the practical software design quideline through the past three years works.

The products of this working are summarized the following documents.

(1) PROCOS User's Manual

(2) PROCOS Design Specification

The meaning of PROCOS is Process Control Oriented Software System

This PROCOS is designed to apply process data processing and control, and to utilize past experience and expected new hardware/software technology. This PROCOS is developed to utilize data base handling and language processing capability, and these capabilities are to add the flexibility and the reliability on software system.

To use PROCOS, the following features are given.

(1) Reduce software development cost

(2) Make accurate and reliable system

(3) Make good documentation

(4) Easy modification

But, there are required more improvements to make more efficiency and compact package. Besides, there are needs to more application oriented functions. To prepare this PROCOS is the first step to make progress in future and to use for education.

2. PROCOS design concepts

2.1 Scope of PROCOS

PROCOS is the software package system to use fill in the form (FIF) for writing application specification and procedure. PROCOS consists of the following twelve forms.

(1) Process Variable Input Processing

(2) Process Variable Monitoring

(3) Process Constant Definition

(4) Process Variable Output Processing

(5) Calculate Processing

(6) DDC Processing

(7) Logical Variable Processing

(8) Status Change Detecting

(9) Digital Output Processing

(10) Timer and Counter Processing

(11) Message Processing

To support these FIF functions, the following standardized procedure are prepared.

(1) Operator's console functions

(2) Task control functions

(3) PROCOS language processor

(4) System generation and Data base preparation

PROCOS exhibits the next capabilities of applications.

(1) Process data gathering and reporting

(2) Calculation of indirect measurement value and process performance

(3) Monitoring and alarming of abnormal process status

(4) Supervisory and set point control

(5) Direct digital control

(6) Sequence control

(7) Man machine communication by process operator's console and message typewriter

(8) On-line system modification and easy system change

(9) System protection and security

2.2 Requirements of computer resource

PROCOS is considered to utilize the following computer capabilities

(1) Host Machine

(1.1) To perform simulation of 16 bits machine

(1.2) To have more than 32K bytes main memory

(1.3) To have more than 512K bytes auxiliary random access memory

(1.4) To use FORTRAN compiler

(1.5) To make easily data base by using index sequential file processing

(2) Target Machine

(2.1) 16 bits machine

(2.2) More than 16K bytes main memory

(2.3) CRT operator's console or console input/output typewriter

(2.4) Required random access memory to store PROCOS data base

(2.5) Required main memory to store data base and program of DDC

2.3 Basic program configuration

The programs of PROCOS are consisted, by many kinds of functional programs that are as follows.

(1) Functional program scheduling program

(1.1) Process variable processing scheduling

(1.2) Process monitoring scheduling

(1.3) Calculate scheduling (PROCOS language executive scheduling)

(1.4) Message and Reporting scheduling

(2) DDC scheduling program

(3) Process variable handling program

(3.1) Input handling

(3.2) Conversion handling

(3.3) Output handling

(3.4) Variable and status editing

(3.5) Process monitoring

(3.6) DDC function

(4) PROCOS language interpreter

(5) Message/Report interpreter

(6) Operator's console function program

(6.1) Operator's console executive

(6.2) Operator's console service function

(7) Data base generation and maintenance program

(8) System initialization program

(9) System diagonostics program

These programs must be developed from the following view points.

(1) Restriction of generalization (Flexibility and Applicability)

(2) Compact design

(3) High response characteristics

(4) Definition of hardware address and name

(5) Inter-linkage with operating system (OS)

(6) Inter-linkage with procedural language

(7) Program development support

(8) Program simulation aids

(9) Machine independence

To make these functional programs of PROCOS by using FORTRAN, there are following points to be settle.

(1) Target code efficiency

(2) Data base facilities

(3) Bit string manipulation

(4) Storage allocation

(5) Task control

(6) Program control

(7) Parallel I/O processing

(8) Adeptability

3. FIF Design

3.1 Purpose

The purpose of the fill in the form (FIF) is easy documentations and one writing program preparation works.

FIF must be designed to satisfy the following requirements.

(1) Easy use for control engineer or system designer, but non-computer engineer.

(2) Select descriptive items that are application oriented.

(3) Easy modification by using operator's console.

(4) Good documentation.

(5) Easy writing user's function by using PROCOS language.

3.2 Use of FIF forms

FIF forms are writen by user or system designer, prepared as card media or paper tape media and stored in the defined area of data base.

The format of FIF are shown in the next page.

The items which are required for internal processing in computer are not specified in the FIF formats.

3.3 Naming of Variable

Variable name is consisted by the six characters and auxiliary code (two characters) that the first character is alphabetic.

EXP: YYZZZA (BB)

If, variable name of digital or logical variable is not given for single bit, this name are given automatically as BI XXXX for digital input and BO XXXX for digital output.

3.4 Use of calculate processing

The form sheets of Calculate are writen by using PROCOS language. These described sheet's contents are condensed and translated in pseudo-interpretive code for easy execution.

3.5 Use of Message and Report writing

The form sheets of Message and Report writing are writen by using following rule.

(a) Designation of line position

P : New page
W : Write in W-th row.
SW : Skip the W-pieces of rows
SO : Print in the following print position
Blank : Change the row and print
(Where W is integer number.)

(b) Designation of character position

W : Print after W-pieces of characters
Blank: Print in the following position

(c) Message

° n X : Blanks (n-pieces)

° n_1 n_2 . . . n_n ; character strings

° % p (d_1 d_2. . . . d_n); Description of process

where p: Process variable name

d_i: Process variable elements

NAME w : Process variable name

Fw.d, Ew.d, Dw.d, Iw: Value

UNIT w ; Engineering unit

STATUS Sw ; Status

(n) Fw.d ; Valve of Auxiliary Code (n)

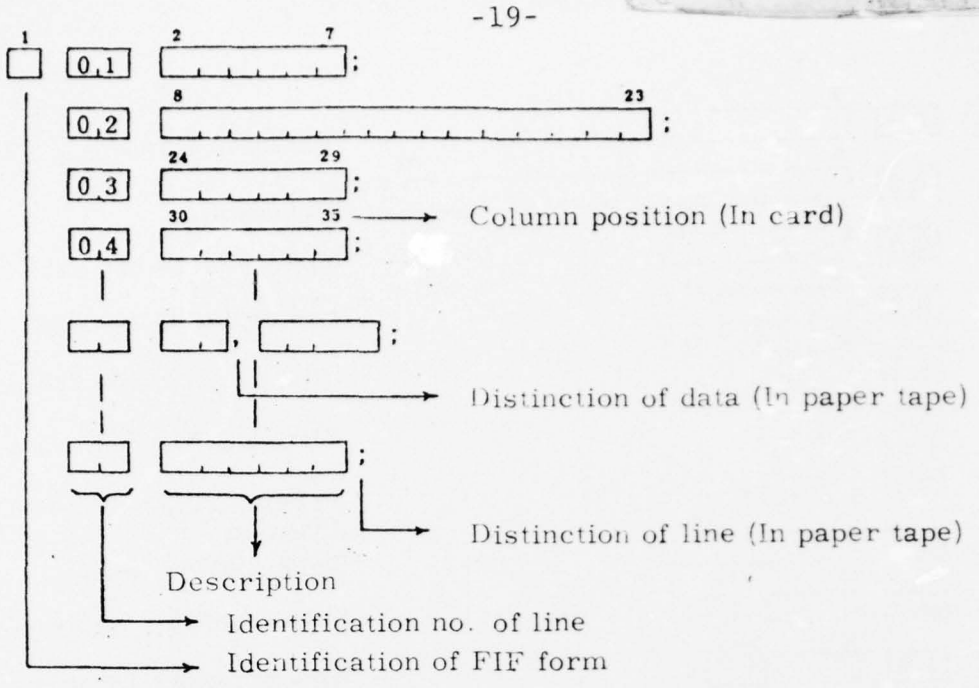
nX; Blanks (n - pieces)

° DATE ; Present date

° TIME ; TIME

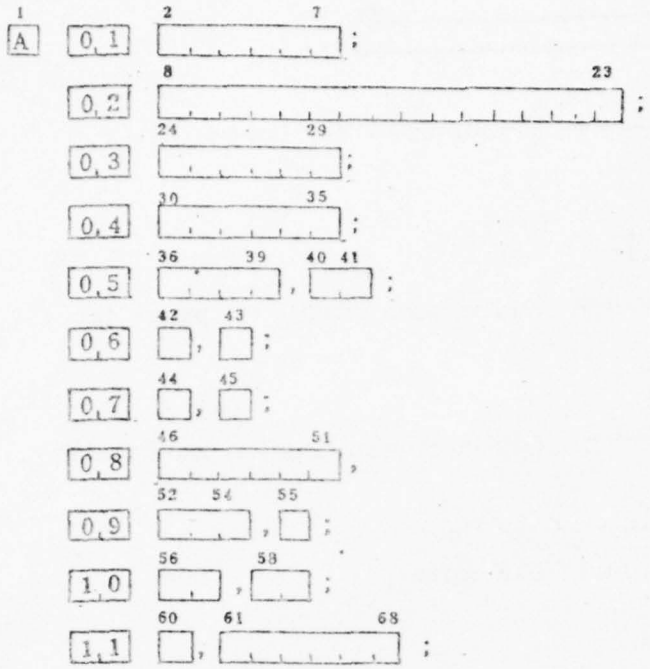
° MESEND ; End of message

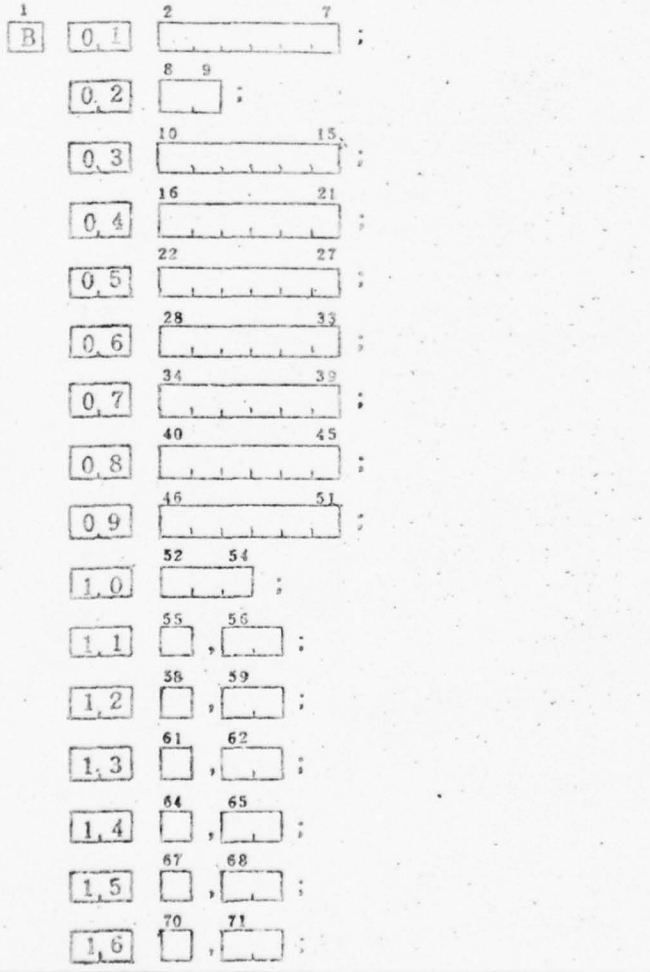
PRECEDING PAGE BLANK-NOT FILMED



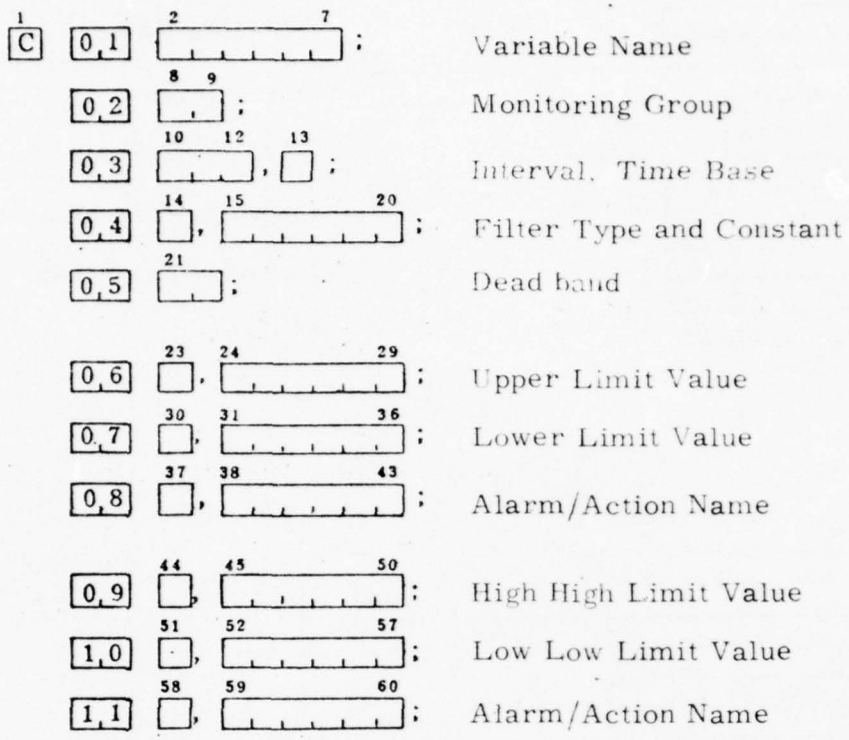
Nortation in FIF form

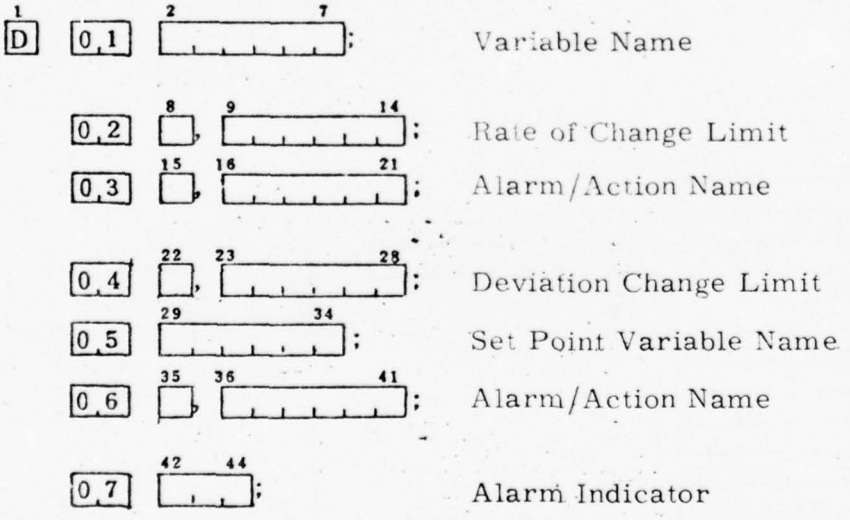
Process Variable Input Processing



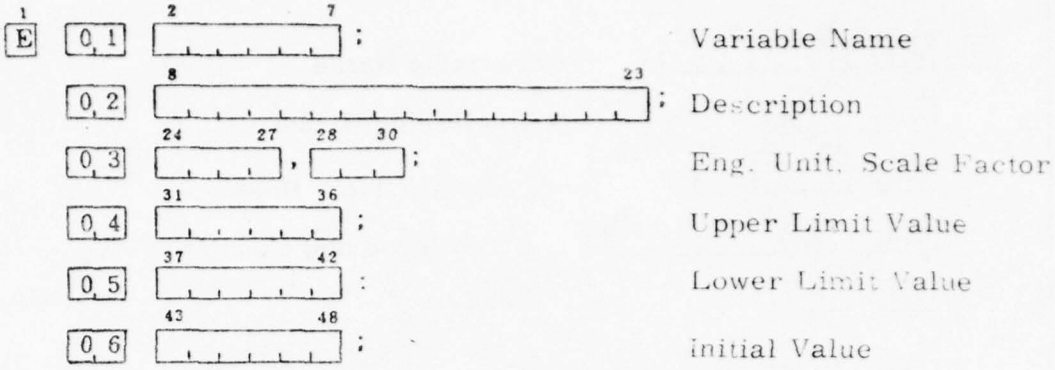


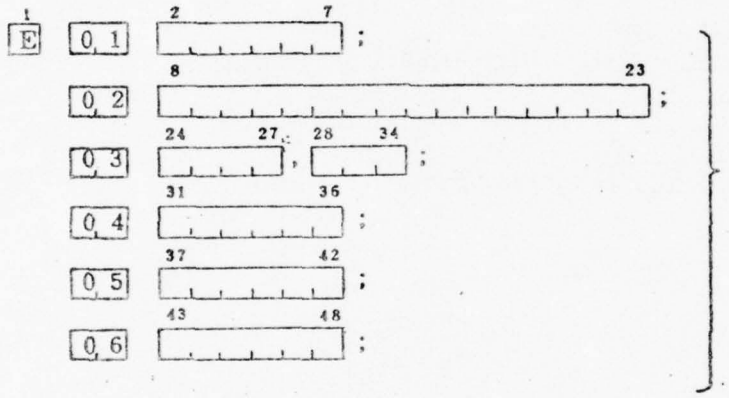
Process Variable Monitoring

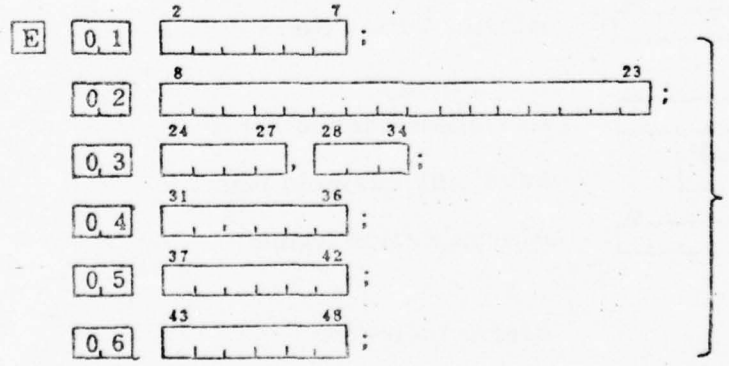




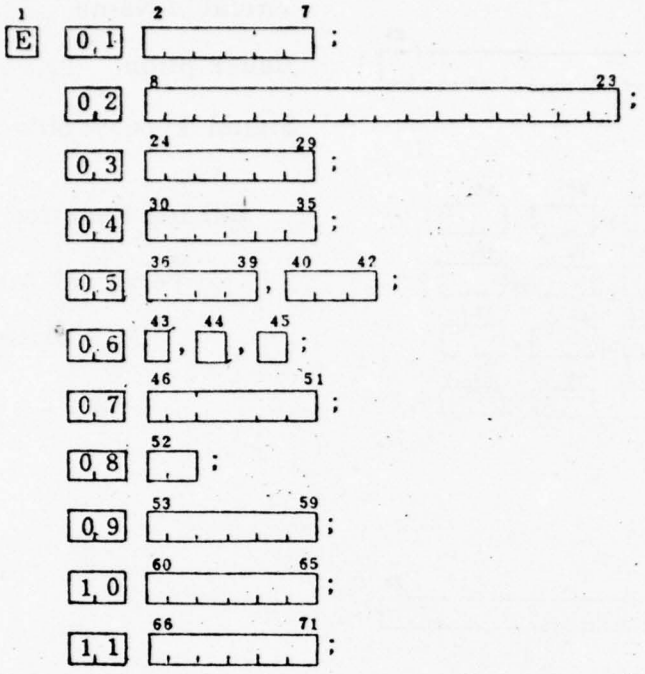
Process Constant

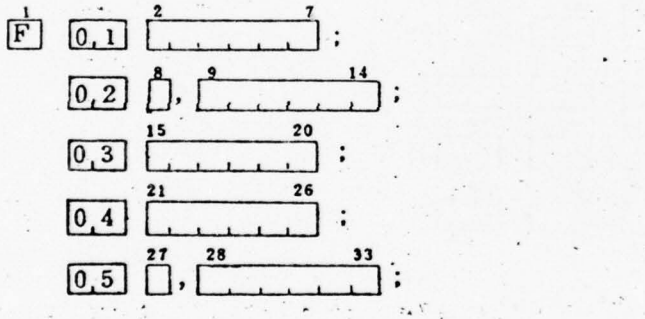




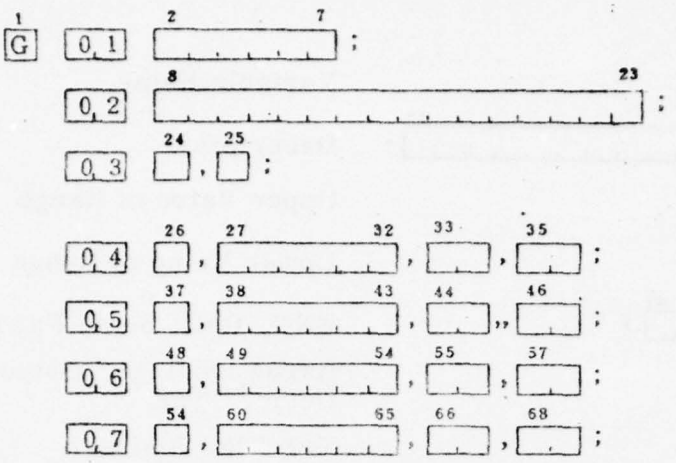


Process Variable Output Processing





Logical Variable Processing



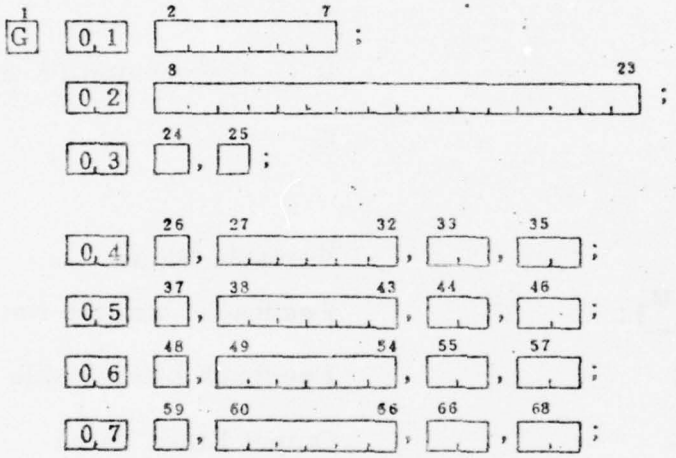
Variable Name

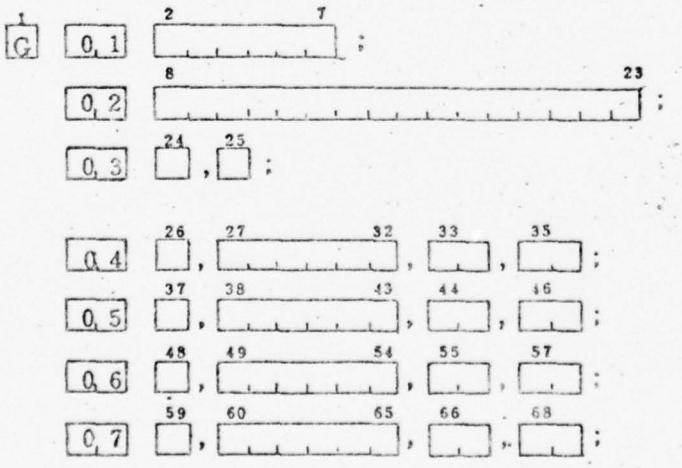
Description

Signal Type, Contact Type

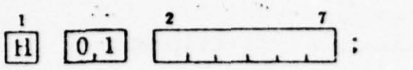
Editing Function

- Terminal Address
- Bit Position
- Bit Length

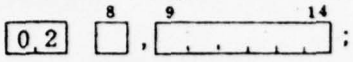




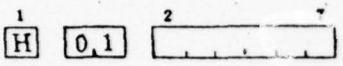
Status Change Detecting

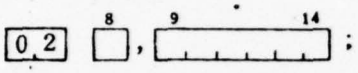


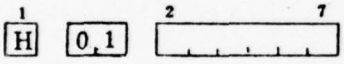
Variable Name

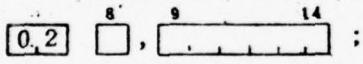


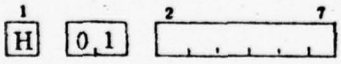
Trigger Condition, Task Name

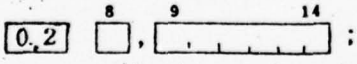


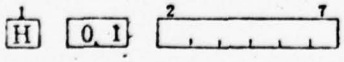


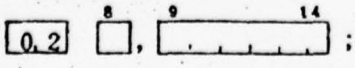




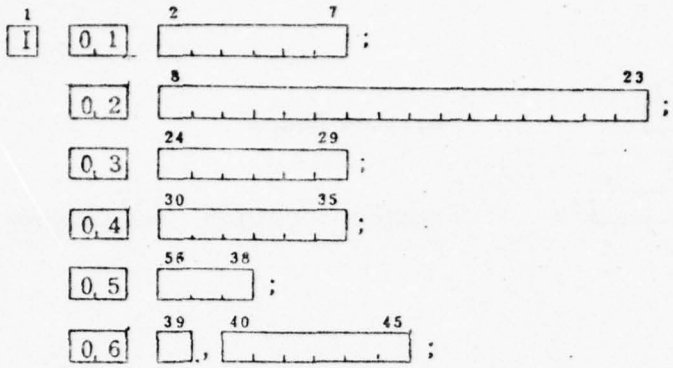


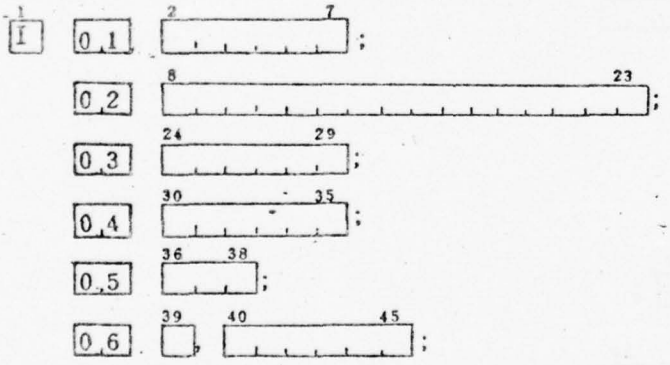




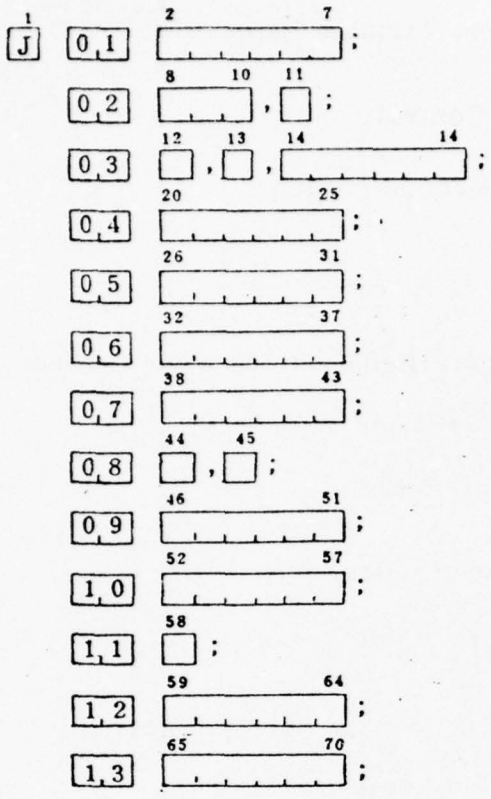


Digital Output Processing

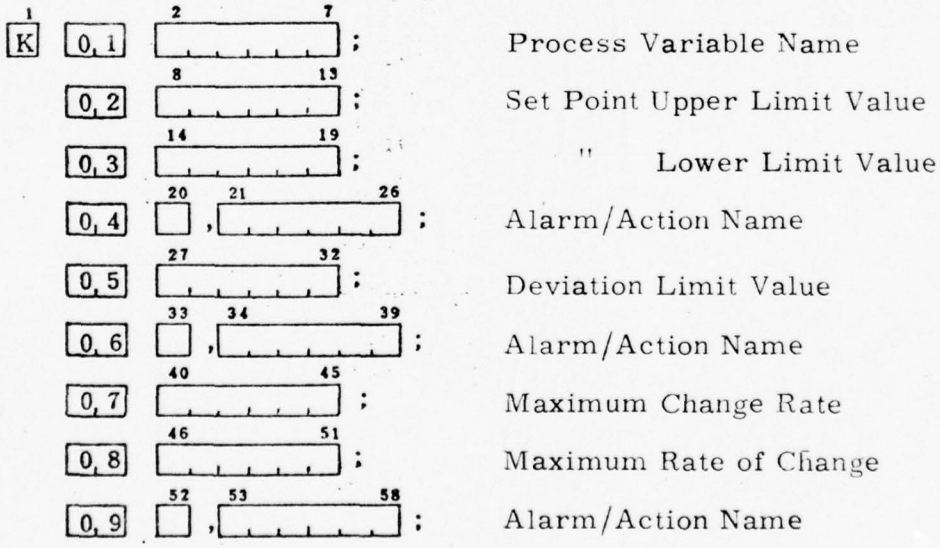




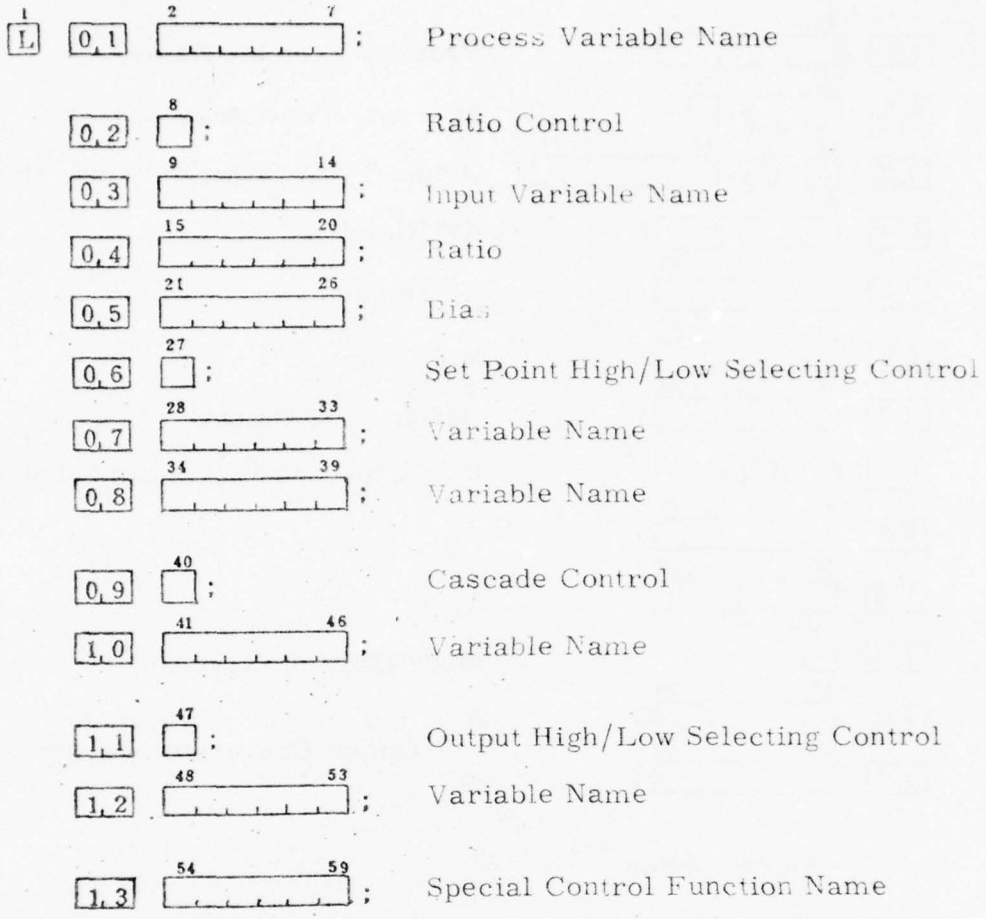
DDC Processing (1)



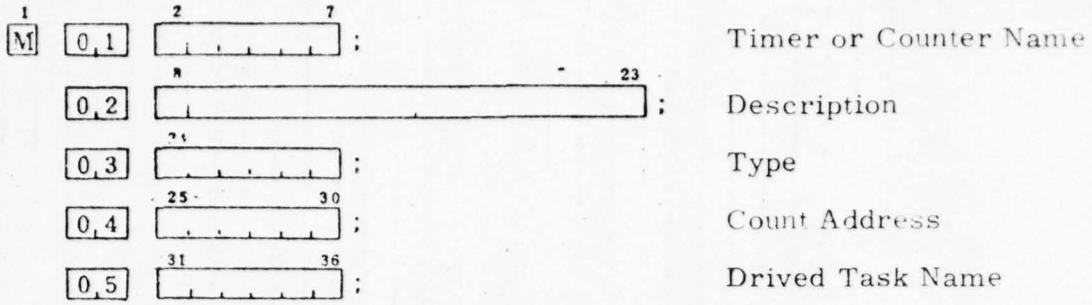
Alarm Option

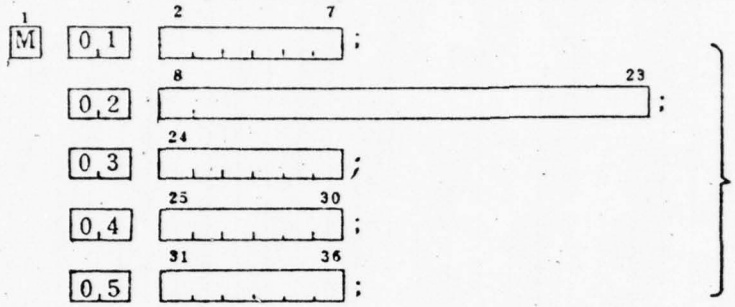


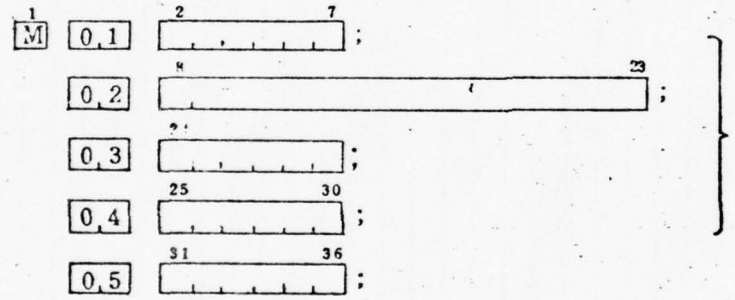
DDC Processing (2)

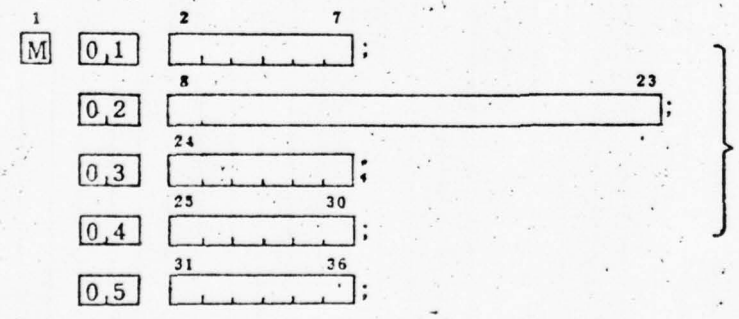


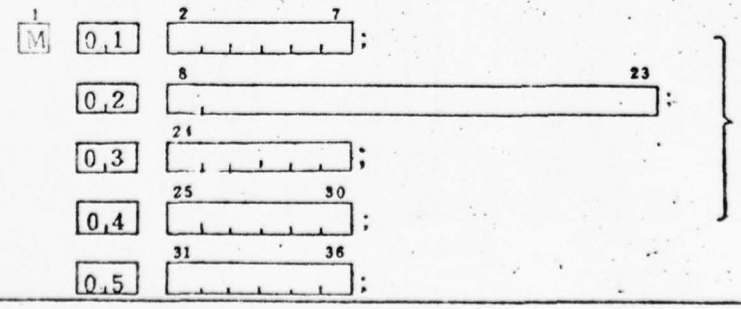
Timer and Counter Processing

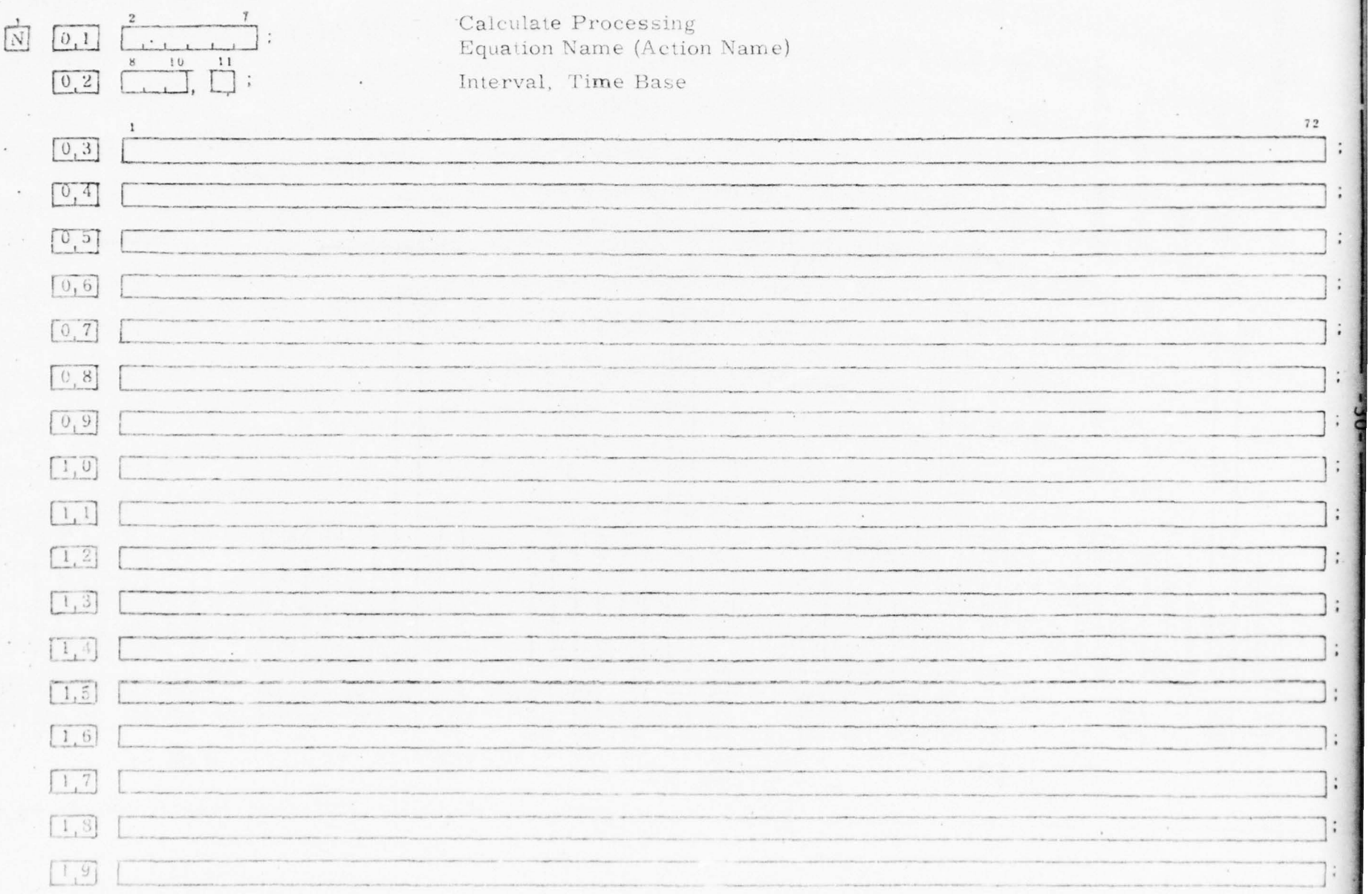


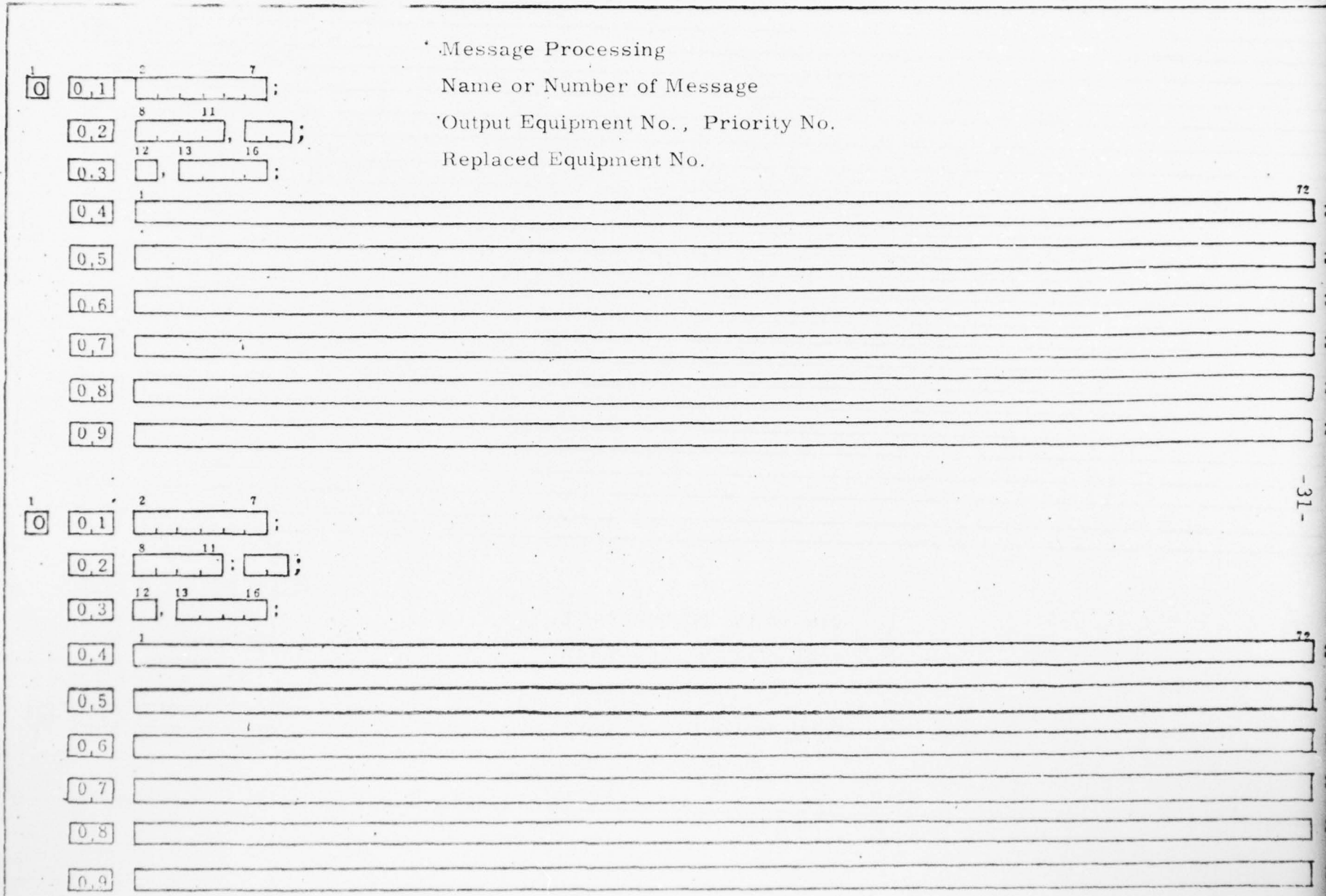


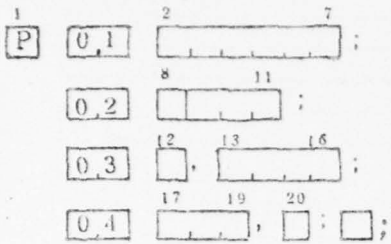







1
N
0,1
2
7
;
Calculate Processing
Equation Name (Action Name)
0,2
8
10
11
,
;
Interval, Time Base
1
72
0,3
0,4
0,5
0,6
0,7
0,8
0,9
1,0
1,1
1,2
1,3
1,4
1,5
1,6
1,7
1,8
1,9


Message Processing
Name or Number of Message
Output Equipment No., Priority No.
Replaced Equipment No.
0
0,1
0,2
0,3
0,4
0,5
0,6
0,7
0,8
0,9



Report Processing

Name or Number of Report

Output Equipment No.

Replaced Equipment No.

Time Interval, Time Base



4. Data base design

4.1 General

The data base of PROCOS system is designed for easy modification and build up, module structure and expandability, and easy documentation.

The access of data base is done by using process variable name which is given at FIF formsheet by user.

The data base at execution phase is consisted as follows and its details are shown the table.

(1) File for Processing procedure

(2) File for Variable value

(3) File for Status and condition

(4) File for Address pointer

(5) File for Executive scheduling

For system generation and preparation, Source Input data by using FIF is filed in Source Data Base.

4.2 Data base access

Data base in PROCOS is refered by using the following procedures.

(1) Variable name

Exp. FRC 101 (PV)
FRC 101 (VP)

(2) Access as vector data or data base block.

Exp. GET (FRC 101)
PUT (FRC 101)

(3) Access from User's FORTRAN program

Exp. CALL GET
CALL PUT

4.3 Data base protection

Data base and its attribute is design to protect destruction.

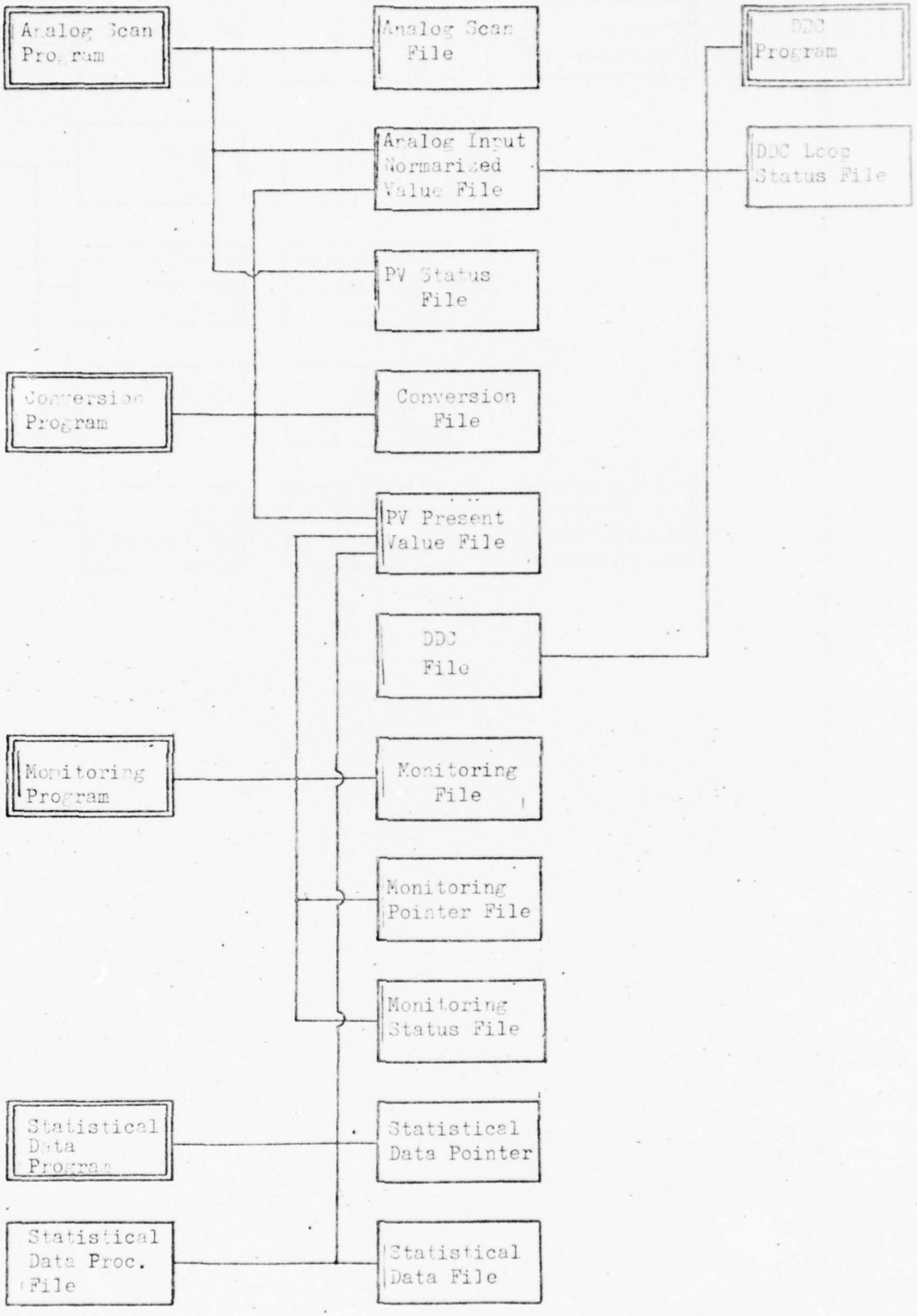
4.4 File structure

Data base is design by using the fixed records file and variable records file for each usages.

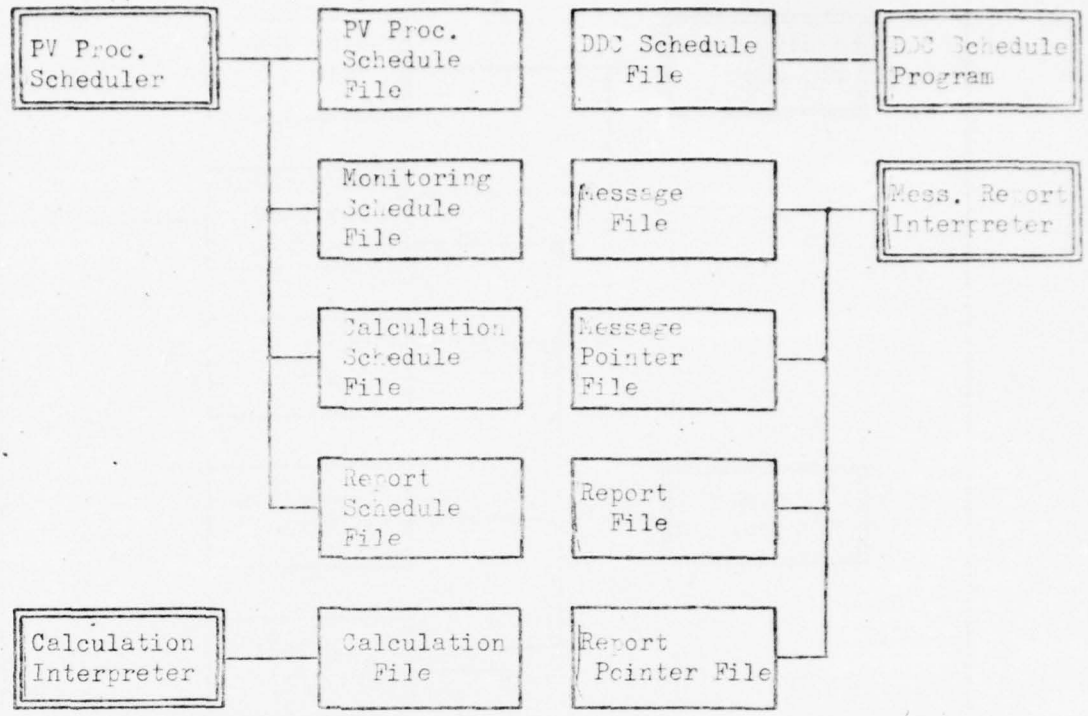
Data Base and Function Classification

Classification	Functions				
	Procedure File	Value File	Status File	Address Pointer File	Scheduler
PV Input Proc.	PV Scan File	PV Normarized File	PV Status		
PV Conversion	Eng. Unit Conv. File	PV Present Value File			PV Scheduler
PV Statistical Data	PV Statisticale File	PV Statisticale Data File		Pointer File Assign File	
PV Monitoring	PV Monitoring File	PV Old Value File PV Set Point File	PV Monitoring status	Pointer File Assign File	PV Monitoring Scheduler
PV Output Proc.	PV Output File	PV Output Value File	PV Output Status		
DDC Proc.	DDC File DDC Scan File	DDC Value File	DDC Status	Pointer File Assign File	DDC Scheduler
Status Change Detect	Status Detect File	DI File			
LV Proc.	LV Processing File	DO File		LV Pointer File	
Timer Counter Proc.	Timer Counter File	Timer Counter File			
Calculate	Calculation File			Calculation Pointer File	Calculate Scheduler
Message Proc.	Message File			Pointer File	
Report Proc.	Report File			Pointer File	Report Scheduler
LV Output Proc.	LV Output File	DI/DO File			

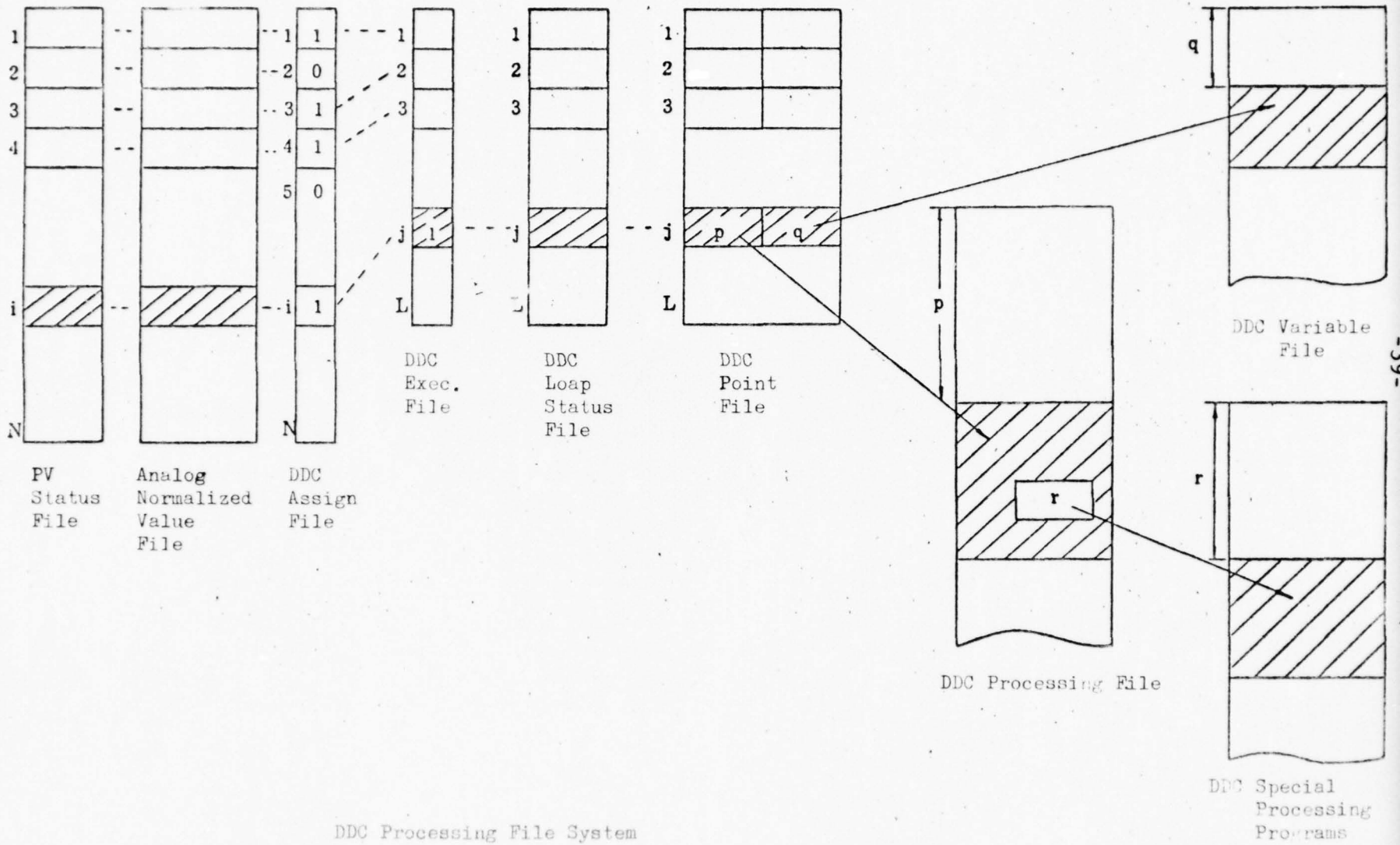
Classification	Functions				
	Procedure File	Value File	Status File	Address Pointer File	Scheduler
Constants		Process Constants File			
Descriptions	Description				
Task Control			Task Exec. Status	Memory Pointer	Task Scheduler
			PROCOS Assign Flag	Task Exec. Address	
File Access	File structure			File start Address File	
	Name index				
				File Record Map File	
				IO File	



Basic Relation between DataBase and Program (1)



Basic Relation between Data Base and Program (2)



DDC Processing File System

PRECEDING PAGE BLANK-NOT FILMED

5. PROCOS language design

5.1 Purpose

PROCOS language is utilized to make user's functional program. This language is designed as reduction of FORTRAN statements.

This language has the functions of arithemetic procedure, bits and logical manipulation, data base input and output handling, simple message handling.

The main applications area are as follows.

(a) User's special logics.

(b) Special compensation and conversion

(c) Advanced control logics

(d) Abnormal function procedure

5.2 Language processing

PROCOS Language is writen in FIF form sheets. The described statements in FIF are converted in intermediate language (Pseudo-code) and stored in data base for execution.

In execution phase, this pseudo-code is interpreted and executed.

5.3 Statements of PROCOS language

(1) Constant/Variable

(a) Real	FORTRAN TYPE	(2 Words)
(b) Integer	"	(1 Words)
(c) Binary	B "101"	(1 Words)

(2) Variable Name

Be based on FORTRAN

(3) Arithemetic operators

Be based on FORTRAN

(+ -, *, /, **, (,))

(4) Relational operators

Be based on FORTRAN

(.EQ., .LT., .LE., .GE., .GT., .NE.)

(5) Logical operators

Logical operations (OR, AND, EOR, NOT) are operated as 16 Bits base.

(6) Operation

Be based on FORTRAN
(By using = (equals)

(7) Test and Branch

Using Logical IF of FORTRAN

(8) Branch

Unconditional GO TO
Computed GO TO

(9) End of Execution (Return to OS)

STOP

(10) End of Statements

END

(11) Keep the area of PROCOS for future extension

SPACE (n)

(12) Subroutine call

Be based on FORTRAN's CALL

(13) Functional and Subroutine subrogram

Not defined

(14) Basic external functions

EXP(x), ALOG(x),
SQRT(x), ABS(x)

N.B. It is possible to add a user's subroutines.

(15) Comments

Be based on FORTRAN

(16) Statement number

Be based on FORTRAN

(17) Message print

PRINT Message no.

(18) Arrays (or Dimention)

Not defined

(19) Status call

Using Macro call

(20) Program drive

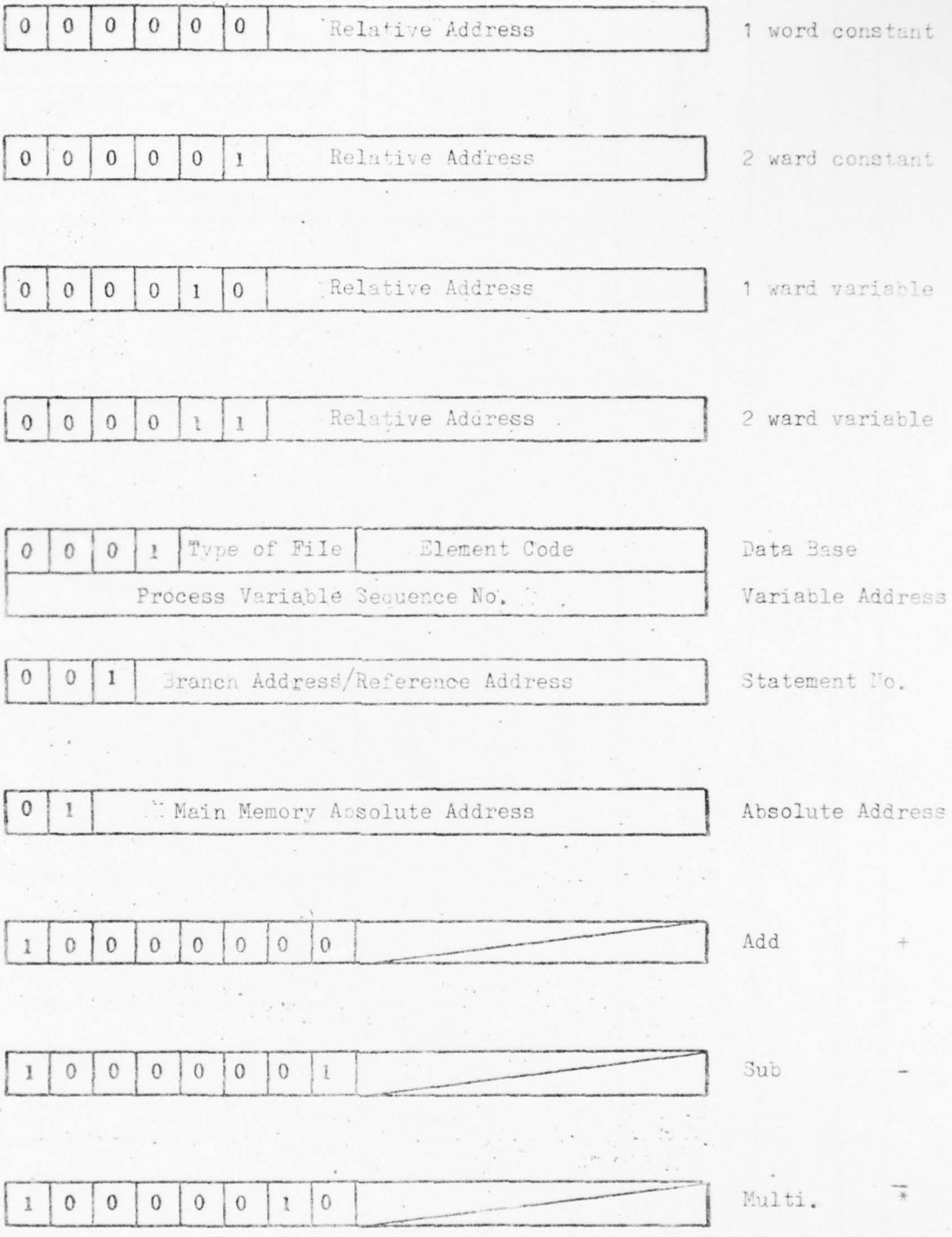
START (i, j, k, m)

(21) Time and Calender

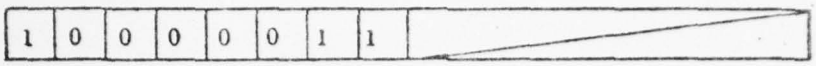
DATE
TIME

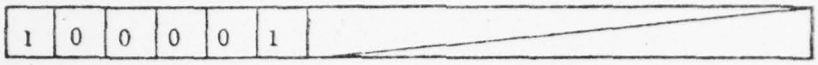
Interpretive Code	Address	Relative	Non-statement Address	Inter Program Address	Constant	1 Word Constant	
						2 Words Constant	
					Variable	1 Word Variable	
						2 Words Variable	
				Inter Data Base Address			
		Absolute Address					
	Instruction	Processing	Processing	Basic Processing	Arithemetic	Basic	Add +
							Sub -
							Multo *
							Div. /
						Power	**
					Non Arithemetic	Logical	.OR.
							.AND.
							.EOR.
							.NOT.
						Relational	
				Function	ABS ( )		
					SQRT ( )		
					Misc.		
			Equation =				
		Branch	Internal Branch	Unconditional GO TO			
				Condi-tional	IF ( ) GO TO		
					GO TO ( ), I		
			External Branch	Sub-routine	CALL		
					CALL GET, CAL PUT		
					MISC.		
				End	STOP		

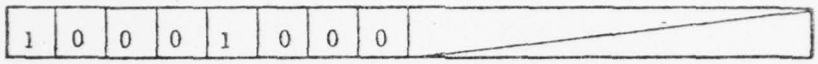
Classification of Interpretive Code

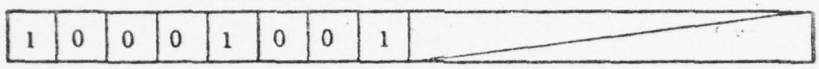


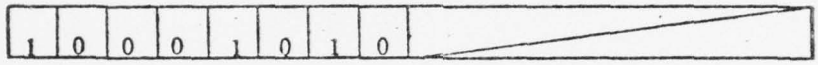
Example; Interpretive Code Assignment (1)

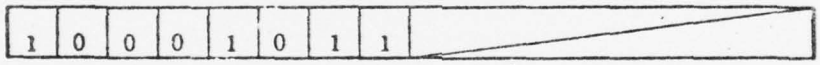


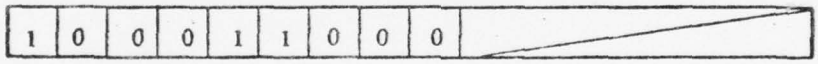


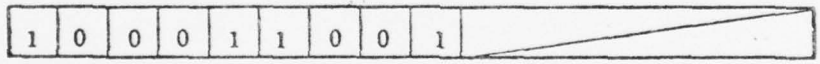


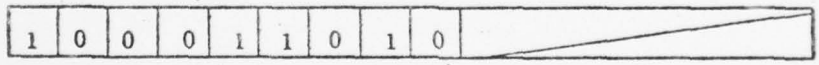


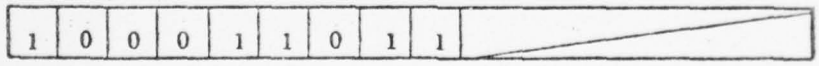




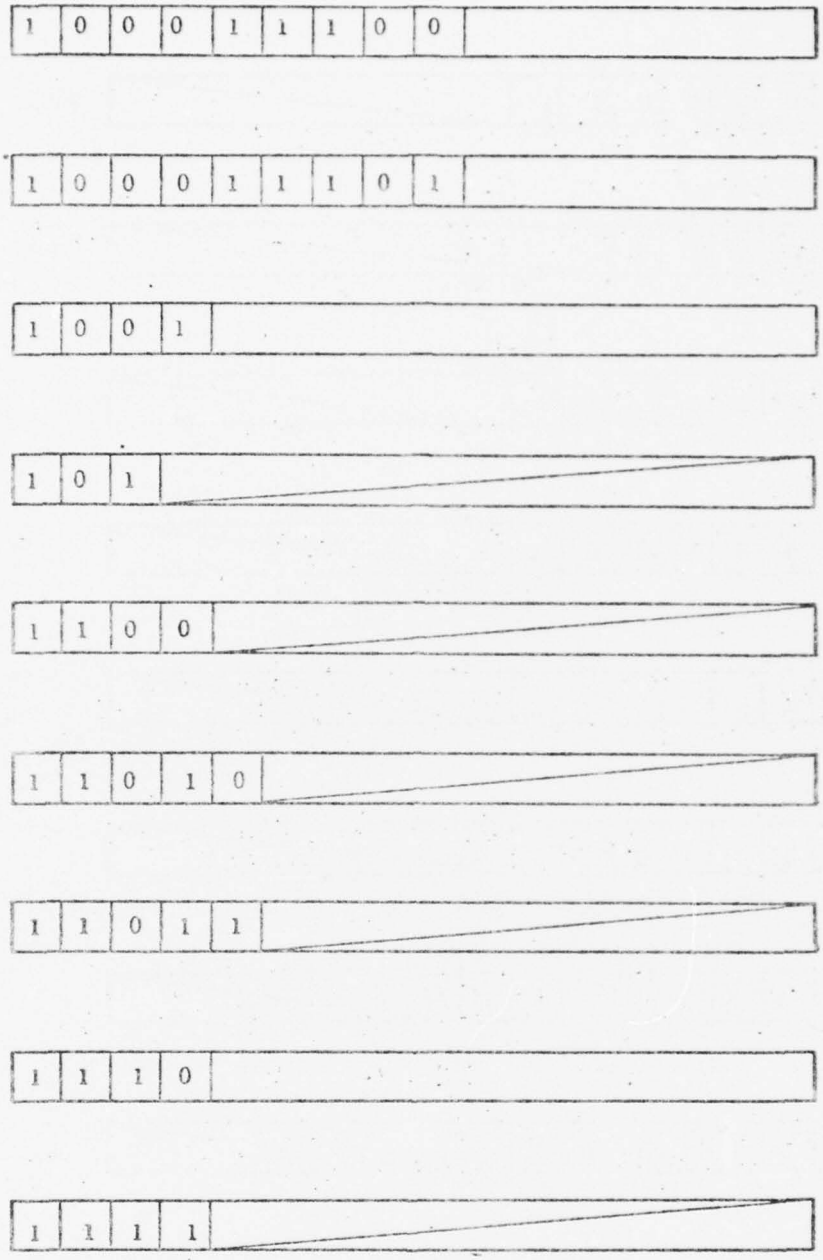








Example; Interpretive Code Assignment (2)


1 0 0 0 1 1 1 0 0
.EQ.
1 0 0 0 1 1 1 0 1
.NE.
1 0 0 1
Function
1 0 1
Equation (=)
1 1 0 0
GO TO
1 1 0 1 0
Logical IF
1 1 0 1 1
Computed GO TO
1 1 1 0
CALL
1 1 1 1
STOP

6. Task control and program design

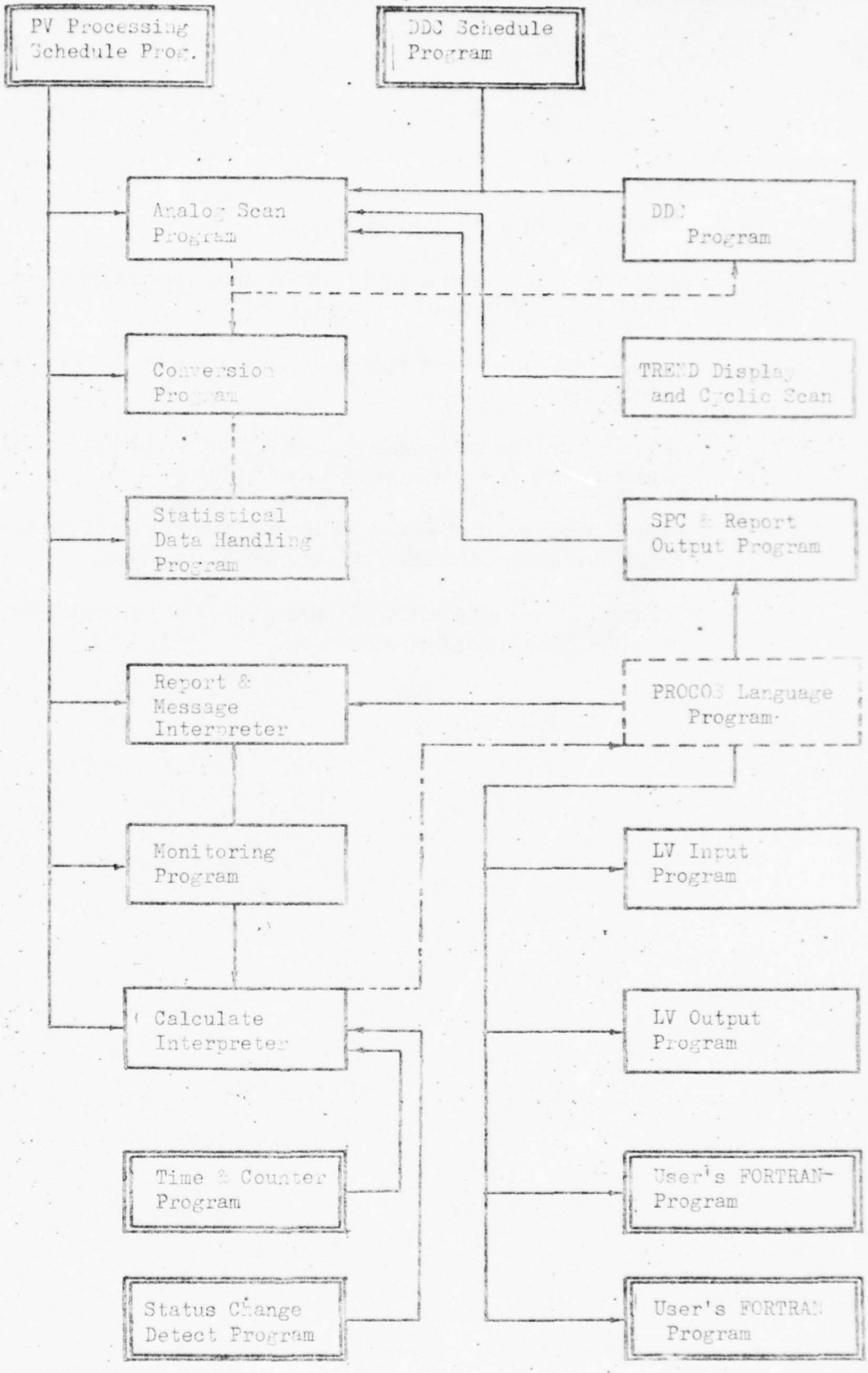
For executive control of PROCOS, the special designed task control functions are prepared.

In general, there are the task control tables to register status and conditions.

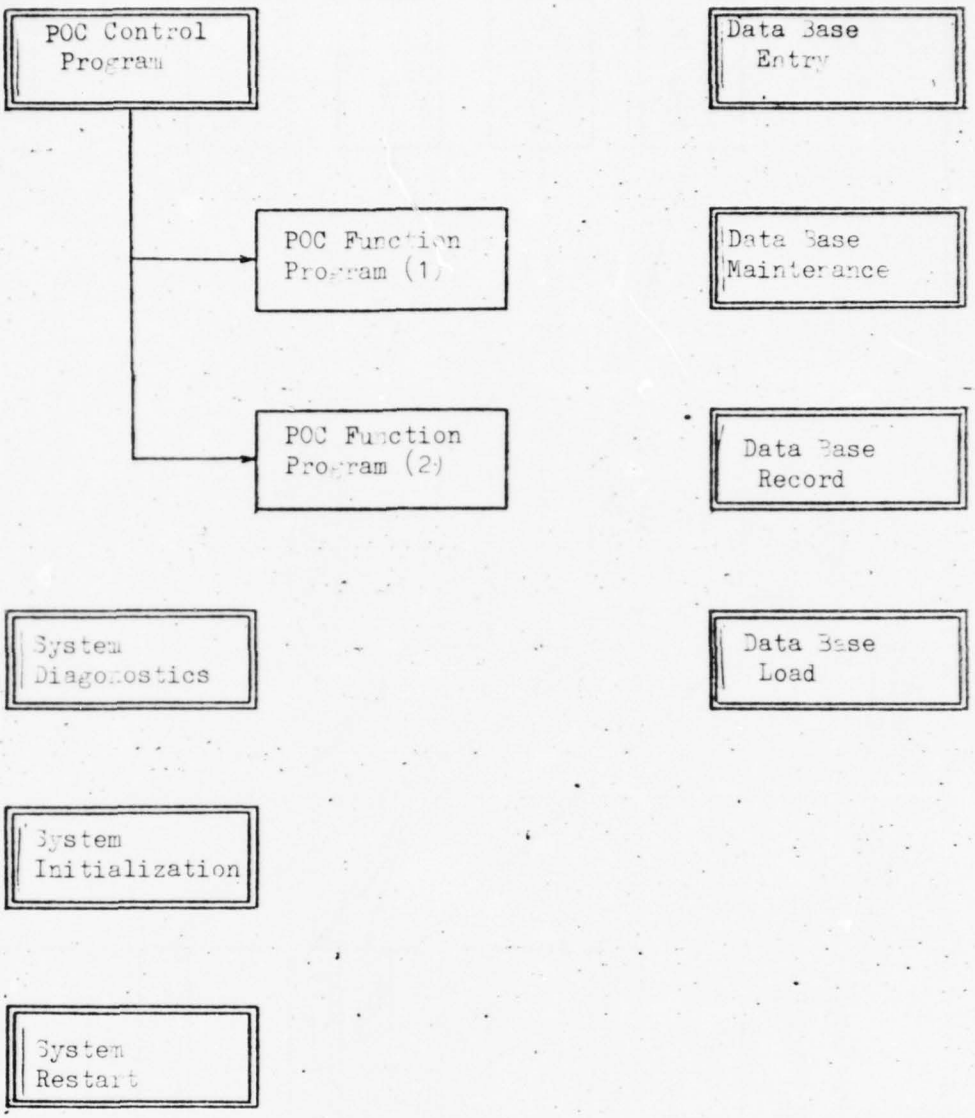
Priority interrupt handling is designed according operating system (OS) and hardware characteristics.

These task control tables have the capabilites of cyclic processing and demand (or request) processing.

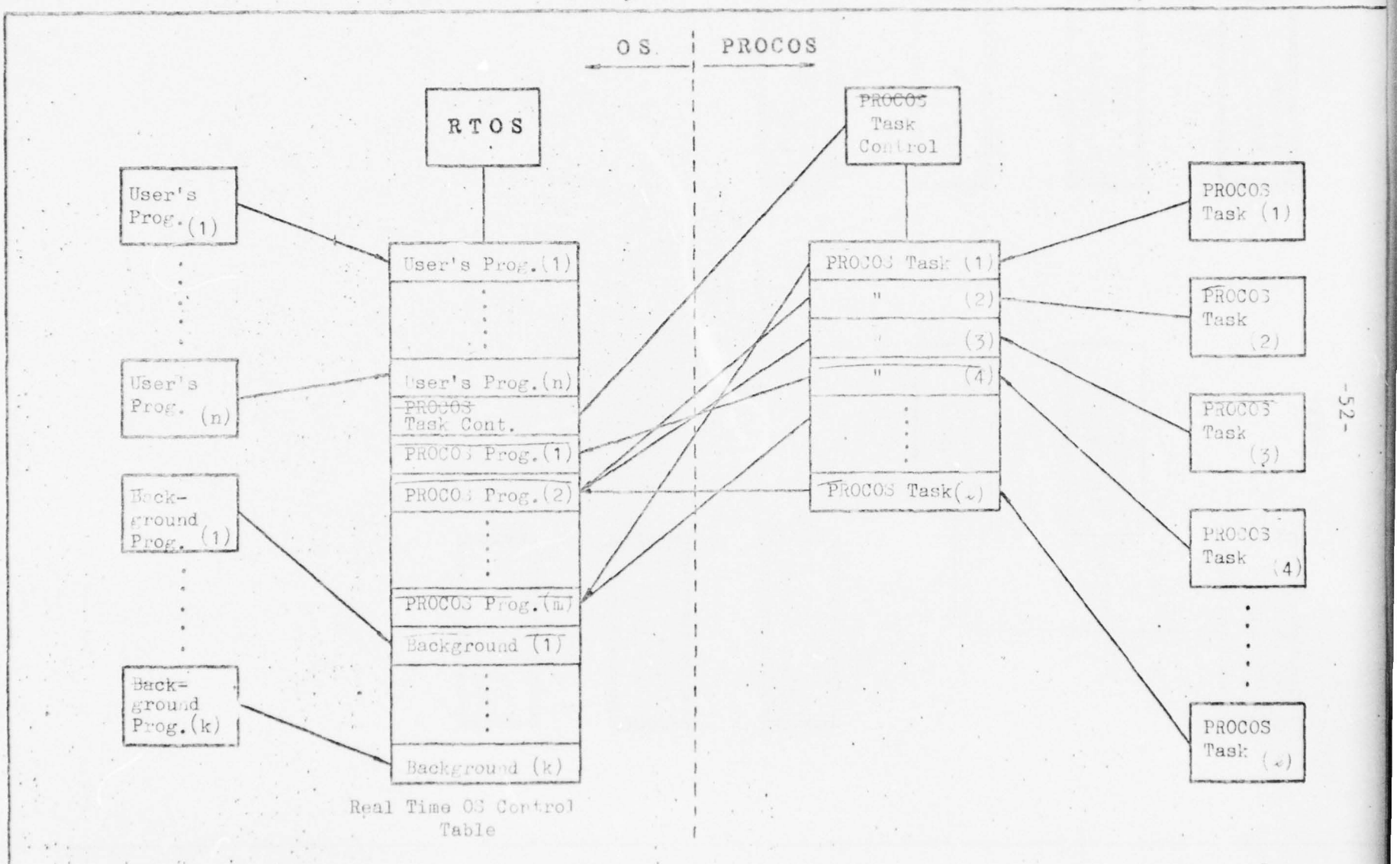
In the following, there are shown the task control relations and the basic program relations.



Basic Configuration of Programs (1)

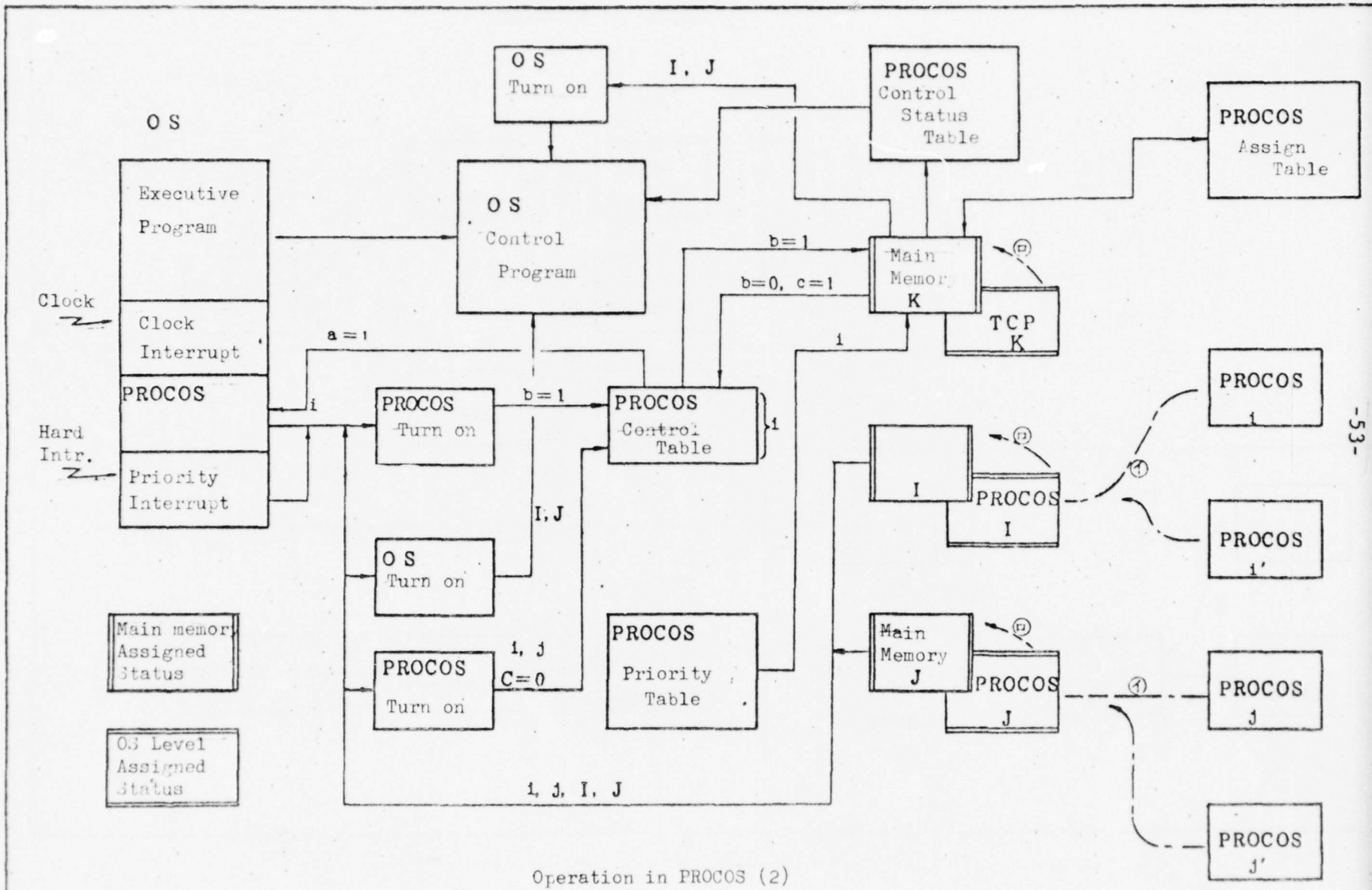


Basic Configuration of Program (2)

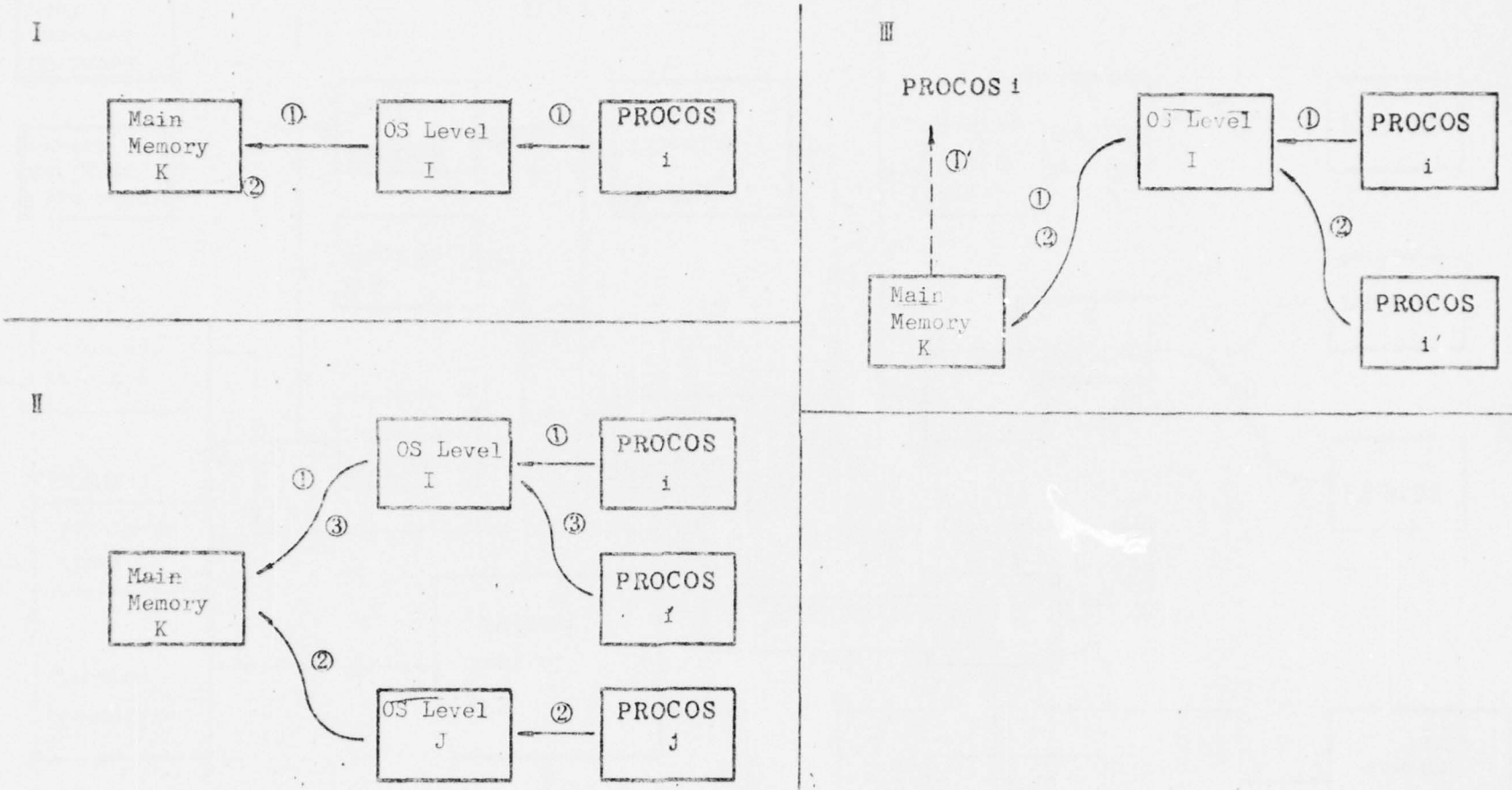


Real Time OS Control Table

Basic Task Control



Operation in PROCOS (2)



Operation in PROCESS (1)

7. Operator's console function design

7.1 General

The operator's console of PROCOS is utilized by using CRT console as character display and data entry system.

7.2 Functions of Operator's console

(1) Process data display (by cyclic display)

- (1.1) One point digital display
- (1.2) Multi point digital display
- (1.3) Multi point bar display
- (1.4) Multi point trend display

(2) Parameter display

- (2.1) Data base parameter display
- (2.2) Control parameter display

(3) Data base page display

(4) Process data setup

- (4.1) One point data set up
- (4.2) Multipoint data set up

(5) Data base parameter set up

(6) Data base condition set up

(7) Data base page set up

(8) Request functions

(8.1) Time display

(8.2) Display request

(8.3) Assign group

(8.4) Report request

(8.5) Alarm summary print

(8.6) I/O signal test

(8.7) Cancel

(9) Alarm display

(9.1) Computer error

(9.2) Computer running status

(9.3) Process monitoring alarm

(9.4) Operator's console manipulation error

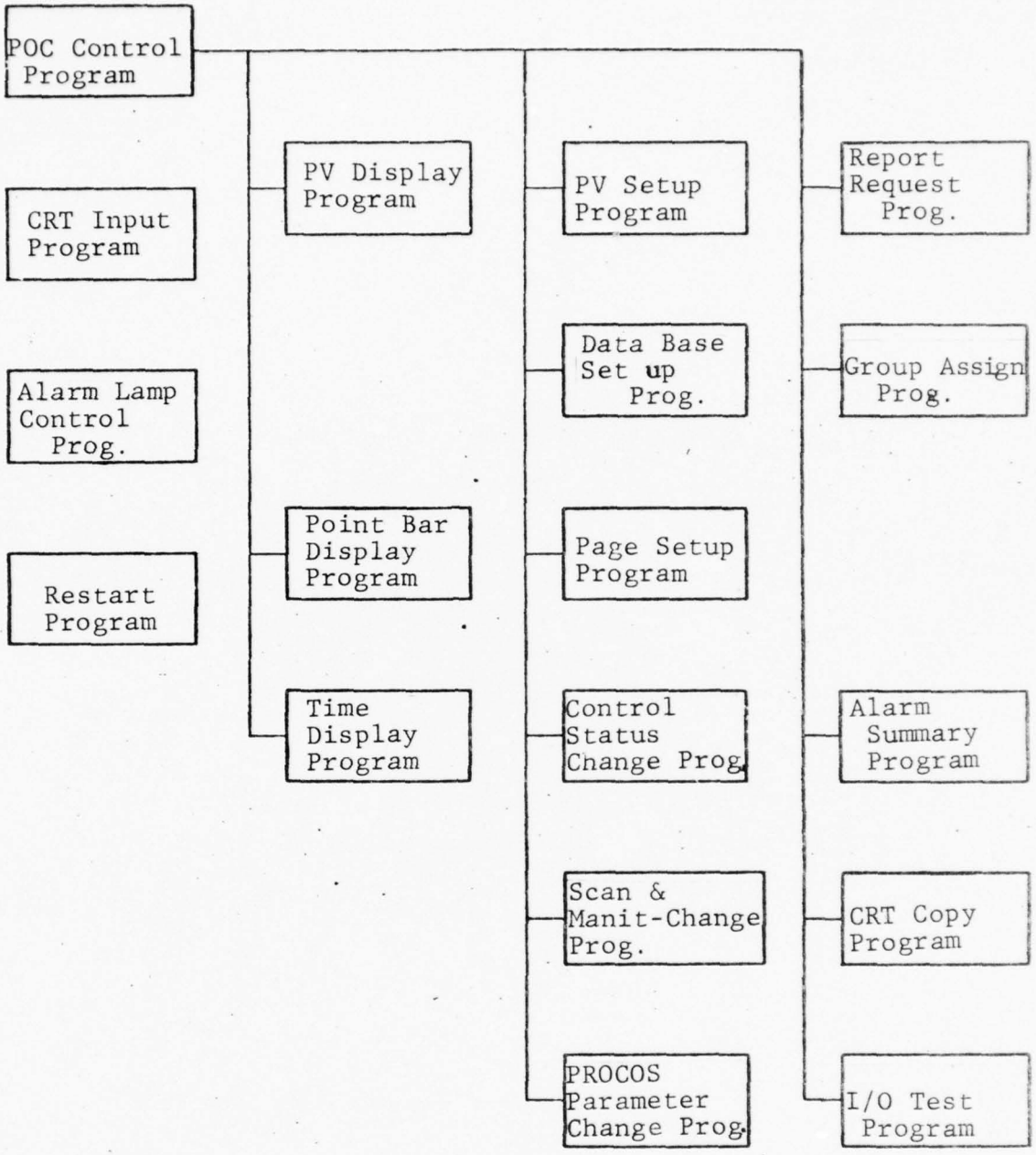
(10) Control loop manipulation

(10.1) DDC loop switching

(10.2) Analog output loop switching

(10.3) Manual output manipulation

(10.4) Loop status display



Operator's Console Program

8. System generation

For flexibility and expandability, the following functions of system generation are prepared.

(1) Data base generation

(2) Program generation

(3) Executive program module generation

(4) Documentation

Basic functions of host computer are as follows.

(1) File creation

(2) File condensation

(3) File modification (or up to date)

(4) Documentation

(5) Program generation

Basic functions of target computer are as follows.

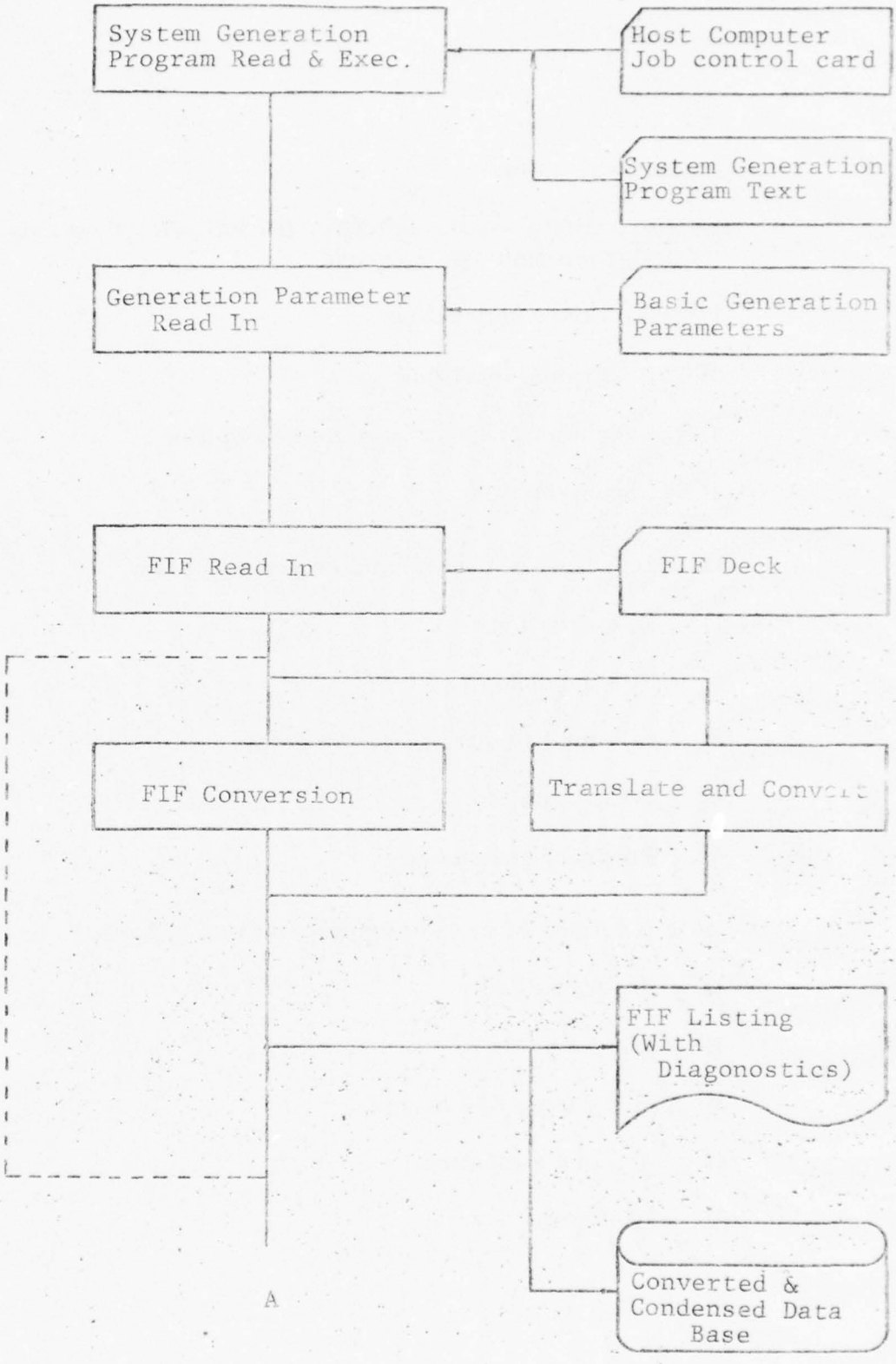
(1) Program load

(2) File load

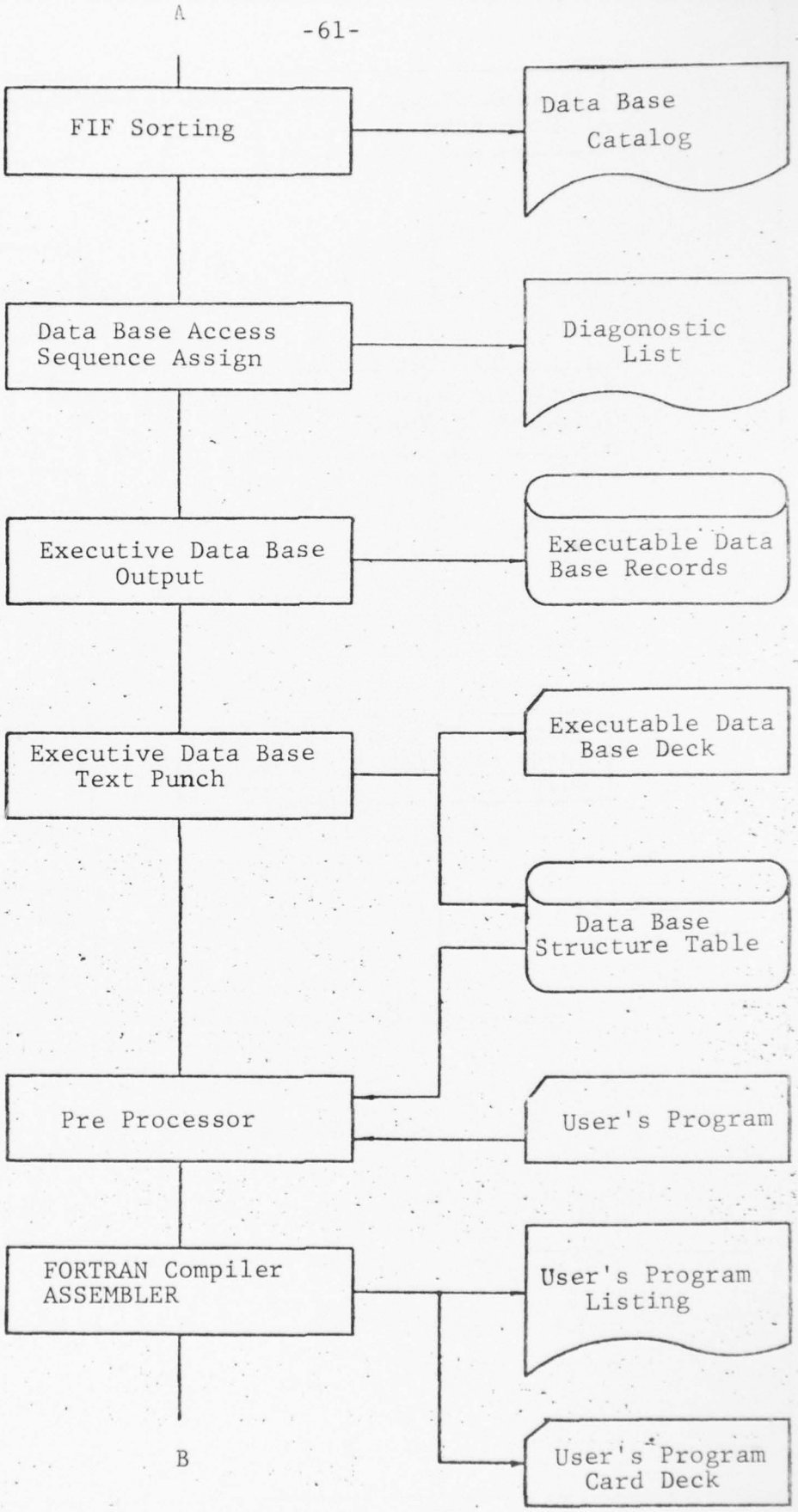
(3) File change and delete

(4) Memory allocation

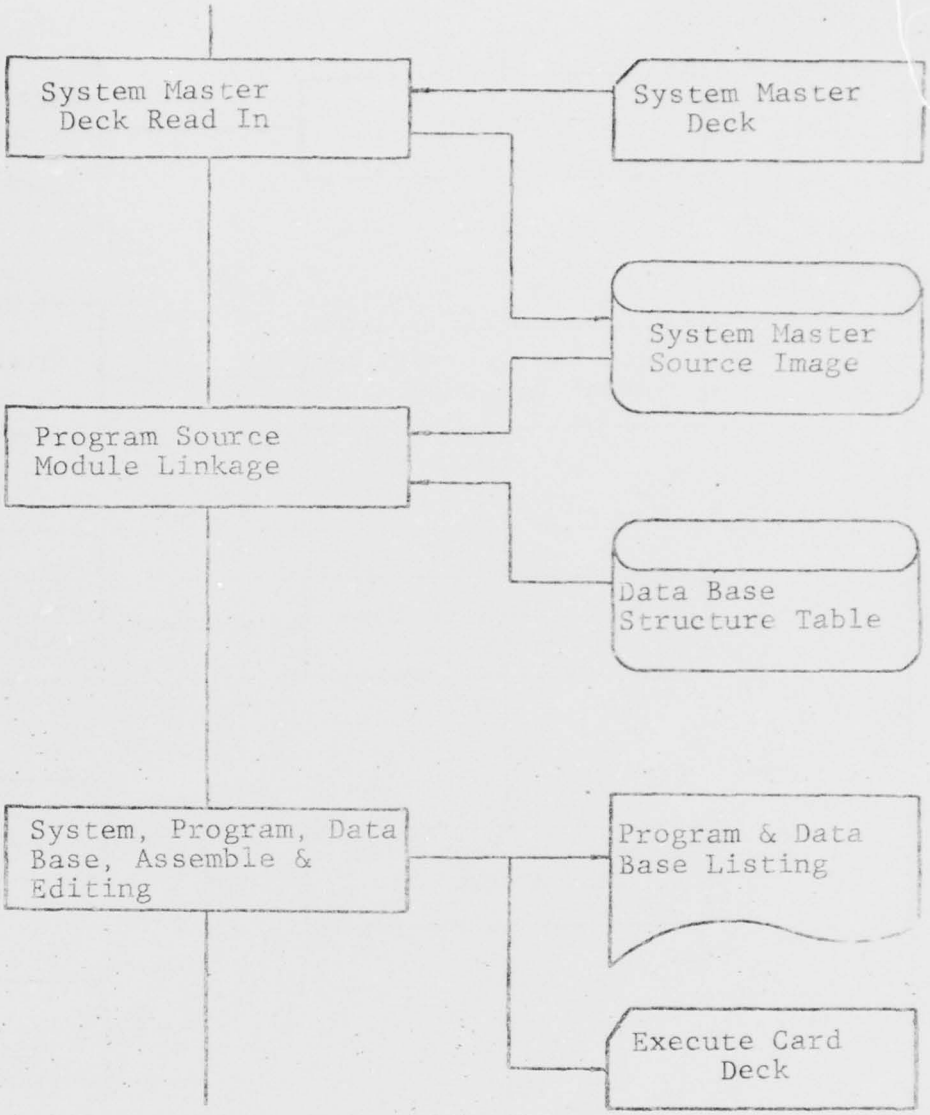
(5) File dump



System Generation (1)



System Generation (2)



System Generation (3)

SECTION II

OUTLINE OF AN EXPERIMENTAL PL/I COMPILER IMPLEMENTATION FOR A MEDIUM SCALE INDUSTRIAL COMPUTER

There has always been considerable interest in both the Japanese and American branches of the Workshop in the possible use of PL/I as the basis for the LTPL. The attached document from the Fourth Minutes of the Purdue Workshop on Standardization of Industrial Computer Languages (pp. 224-232) describes one such implementation carried out in Japan.

PRECEDING PAGE BLANK-NOT FILMED

OUTLINE OF AN EXPERIMENTAL PL/I COMPILER IMPLEMENTATION FOR A MEDIUM SCALE INDUSTRIAL COMPUTER

M. SUDO

1. INTRODUCTION

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 number of new schemes in practical implementation of compilers.

We had an experience to implement a PL/I subset compiler for a medium scale computer prior to designing the compiler for another larger machine. The computer used in this experiment is a medium scale industrial computer with 16 KW core memory and with some rotating type bulk memory devices, which happens to be the same as the basic assumptions of the workshop report refers. The main objective of our experimental implementation was not to develop a practical compiler for that machine but to investigate the appropriate compiling techniques for those functions which PL/I newly adopted.

Although our compiler, therefore, does not suggest directly to the design of procedural languages for industrial use, there may be some implicative suggestions to the definition of PL/I like procedural languages for medium scale computers, especially in case of defining the language through selecting a practical subset from the full PL/I specification.

2. Language Specification

2.1 PROGRAM ELEMENTS

PL/I	Experimental Subset
2.1.1 Basic Language Structure	
a. Language Character Sets	
60-character Set	
48-character Set	48-character Set
b. Delimiters	
Operators	
Parentheses	
Separators	
c. Data Character Set	
d. Collating Sequence	
e. Identifiers	
the maximum length of identifiers is implementation defined	the maximum length of identifiers is 31
f. Keywords	
Keywords are not reserved words	Keywords are not reserved words except the operators (AND, OR, NOT, GT, NG, GE, NE, LE, LT, NL)
g. Use of Blanks and Comments	Same as PL/I
2.1.2 Basic Program Structure	Same as PL/I
a. Simple Statements	
b. Compound Statements	
c. Groups	
d. Blocks	
Procedure Blocks	
Begin Blocks	
e. Program	

PL/I	Experimental Subset
2.2 DATA ELEMENTS	
2.2.1 Data Organization	
a. Scalar Data	
b. Data Aggregates	Structures are prohibited.
Arrays	Upper- and lower bound are defined
Structures	with constants.
Arrays of Structures	
2.2.2 Naming	
a. Simple Name	Simple Name
b. Subscripted Name	Subscripted Name
Cross section of array	
c. Qualified Name	
d. Subscripted Qualified Name	
2.2.3 Data Types	
a. Problem Data	
arithmetic data	complex is prohibited
real/complex	
fixed/float	
binary/decimal	
string data	Length of string is defined with
bit/character	constants.
fixed length/varying length	
b. Program-Control Data	
label data	label data
entry data	
task data	
event data	
locator data	
area data	

PL/I	Experimental Subset
2.3 DATA MANIPULATION	
2.3.1 Scalar Expressions	Same as PL/I
2.3.2 Aggregate Expressions	Prohibited
2.4 DATA DESCRIPTION	
2.4.1 Declaration	Same as PL/I
2.4.2 Attributes	
a. Problem Data Attributes	
REAL/COMPLEX	
BENARY/DECIMAL	BINARY/DECIMAL
FIXED/FLOAT	FIXED/FLOAT
Precision	Precision
BIT/CHARACTER	BIT/CHARACTER
Length	Length
VARYING	VARYING
PICTURE	
b. Control Data Attributes	
LABEL	LABEL
ENTRY	
TASK	
EVENT	
POINTER	
OFFSET	
AREA	
c. Entry Attributes	
ENTRY	ENTRY
VARIABLE	
RETURNS	RETURNS
REDUCIBLE/IRRUDUCIBLE	
GENERIC	
BUILTIN	
OPTIONS	

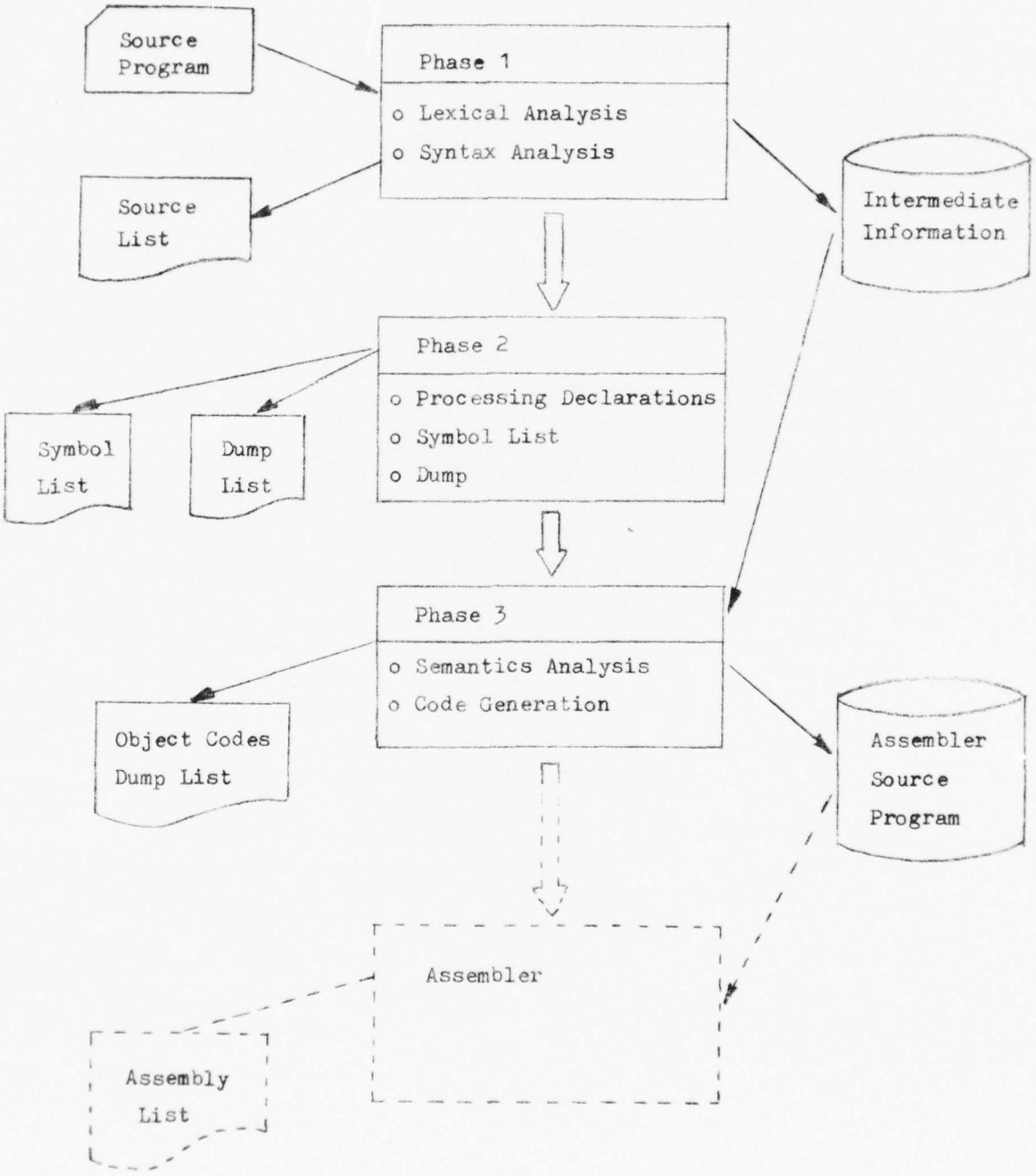
PL/I	Experimental Subset
d. File Attributes	
STREAM/BITSTREAM/RECORD	
INPUT/OUTPUT/UPDATA	
PRINT	
SEQUENTIAL/DIRECT/TRANSIENT	
BUFFERED/UNBUFFERED	
BACKWARDS	
ENVIRONMENT	
KEYED	
EXCLUSIVE	
e. Scope Attributes	
INTERNAL/EXTERNAL	INTERNAL/EXTERNAL
f. Storage Class Attributes	
STATIC/AUTOMATIC/CONTROLLED/BASED	STATIC/AUTOMATIC
g. Others	
ALIGNED/UNALIGNED	
CONNECTED	
DEFINED	
INITIAL	
LIKE	
POSITION	
Parameter	Parameter
Dimension	Dimension
2.5 INVOCATION OF PROCEDURES	
2.5.1 Procedure References	Same as PL/I
2.5.2 Argument Passing	
a. Call by Name	
b. Argument	
{ Scalar argument { Aggregate argument	Aggregate argument is prohibited.

PL/I	Experimental Subset
2.5.3 Generic Names	Prohibited
2.5.4 Built-in Functions	
2.5.5 RECURSIVE Option	Prohibited
2.6 Dynamic Program Structure	
2.6.1 Block Control	Same as PL/I
2.6.2 Storage Control	
a. Static storage class	Static storage class
b. Automatic storage class	Automatic storage class
c. Controlled storage class	
d. Based storage class	
2.6.3 Multi-Tasking	Prohibited
2.6.4 Interrupt Operations	Prohibited
2.7 Input/Output	
2.7.1 Opening and Closing Files	I/O functions are implemented by subroutine reference.
2.7.2 Data Stream Transmission	
2.7.3 Record Transmission	
2.8 Statement	
a. Assignment Statement	Assignment Statement
b. Control Statements	
GO TO, IF, DO, CALL, RETURN, WAIT, STOP, EXIT, DELAY, Null	GO TO, IF, DO, CALL, RETURN, Null
c. Data Declaration Statements	
DECLARE, DEFAULT	DECLARE
d. Error Control and Debug Statements	
ON, SIGNAL, REVERT	

PL/I	Experimental Subset
e. Input/Output Statements	Implemented by subroutine reference.
OPEN, CLOSE, DELETE, UNLOCK, GET, PUT, FORMAT, READ, WRITE, REWRITE, LOCATE, DISPLAY	
f. Program Structure Statements	
PROCEDURE, BEGIN, END, DO, ENTRY	PROCEDURE, BEGIN, END, DO, ENTRY
g. Storage Allocation Statements	
ALLOCATE, FREE	
h. The Others	
FETCH, RELEASE, INCORPORATE	
2.9 Compile-time Facilities	Not implemented.

3. Compiler Organization

3.1 System Flow



3.2 Compiler Size

Phase	Process	Steps	Words	
1	Lexical Analysis	861	1246	7635
	Syntax Analysis	4294	6389	
2	Declarations	896	1651	3563
	Dump	1015	1912	
3	Semantics Analysis	3460	5678	8425
	Code Generation	2461	2747	
Common Subroutines		589	635	

PRECEDING PAGE BLANK-NOT FILMED

SECTION III

STANDARD SOFTWARE FOR CAMAC-COMPUTER SYSTEMS

The developers of the CAMAC standard hardware systems for nuclear instrumentation (see Section I, Part III of this Summary) have also been active in the development of macro-assembly codes for use with this equipment. This Section presents two documents describing this "language."

The first document from the Minutes of the Eighth Meeting of the Purdue Workshop on Standardization of Industrial Computer Languages (Insert VIII-2, pp. 89-99) and the second from the Minutes of the Ninth Meeting (Appendix III, pp. 179-184) present the United States and European recommendations concerning this.

LA-5059-MS
Informal Report
UC-28 and 32
ISSUED: October 1972



los alamos
scientific laboratory
of the University of California
LOS ALAMOS, NEW MEXICO 87544

Specifications for CAMAC Subroutines

by

Richard F. Thomas, Jr.

This work funded by the U. S. Atomic Energy Commission's Division of Research.

SPECIFICATIONS FOR CAMAC SUBROUTINES

Richard F. Thomas, Jr.

ABSTRACT

This report has been prepared under the guidance and sponsorship of the CAMAC Software Working Group of the U. S. AEC NIM Committee. It contains descriptions of six FORTRAN-compatible subroutines recommended for use with CAMAC systems. These subroutines permit the most general operations on a CAMAC system to be specified in a high-level programming language. Two of the subroutines implement the fundamental CAMAC functions; three of them implement the three uses of the Q response defined in TID-25875; the remaining subroutine provides a flexible alternative for CAMAC addressing.

The NIM-CAMAC Software Working Group has agreed that the specification of a standard set of subroutines for performing CAMAC operations would be of use in facilitating communication, permitting interchange of programs and saving of design effort in the implementation of CAMAC systems. The subroutines should be applicable to as many languages as possible, but must be compatible with FORTRAN, which is the most widely used high-level language in the U. S. at this time. This report describes six subroutines which the Working Group recommends for standard usage. Other subroutines are under active consideration, but it is felt that the best interests of the CAMAC user community are served by rapid publication of the specifications which have been agreed upon to date.

The approach which has been followed in the following specifications is that a single subroutine, "CMCBSC," capable of performing a very general CAMAC operation, is specified in terms of its inputs and outputs. Then other, more specialized, subroutines are defined as FORTRAN subroutines which make use of the fundamental subroutine, "CMCBSC." It should be especially noted that although all subroutines except "CMCBSC" are defined as FORTRAN subroutines, an individual installation will quite likely find it advantageous to implement many of them in assembly language, or perhaps in other special ways, in order to take advantage of local hardware and speed up their execution. In these cases, the FORTRAN subroutine will serve only as a definition. However, any installation which can provide a local version of "CMCBSC" and a FORTRAN compiler will have immediately available a library of CAMAC procedures.

The following general remarks apply to the descriptions of all the subroutines which follow. In specifying subroutine arguments names of arguments which may be or must be arrays end with the letter "A," and all arguments are integer variables in the FORTRAN sense. All subroutines, function and named common names, i.e., all global names beginning with the letters "CMC" are reserved for possible future extensions of the standard subroutines. To the maximum extent practical in the specifications argument names and meanings are identical from one subroutine to another. Certain arguments (especially crate number, repeat count, data block and error vector) are explained in detail in the description of subroutine "CMCBSC" and only briefly in the definitions of subsequent subroutines. However, the full definition is intended to apply in every case.

It is necessary for many of the subroutines to have access to the values of certain constants which describe the system, in particular the word length of the computer, the number of modules per crate, and the number of crates per branch. In order that the

subroutines can be written in a general way and be reasonably independent of the actual values of these quantities, a named common which contains the values for the system is defined, a sample "BLOCK DATA" subroutine is given in Fig. 1 for a 16-bit computer with normal values for the quantities.

```
      BLOCK DATA
      COMMON/CMCCOM/NCOMP,NOMOD,NOCRT
C
C       STANDARD NAMED COMMON FOR CAMAC SUBROUTINES.
C
C               NCOMP: NO. OF COMPUTER-LENGTH WORDS NECESSARY TO CONTAIN ONE
C                      CAMAC-LENGTH WORD.
C               NOMOD: NO. OF MODULES (MAXIMUM) IN A CRATE.
C               NOCRT: NO. OF CRATES PER BRANCH.
C
      DATA NCOMP, NOMOD,NOCRT/2,23,7/
      END
```

Fig. 1

I. CMCBSC

Purpose

To perform a single CAMAC function at a single CAMAC address one or more times.

Calling Sequence

CALL CMCBSC(F,B,C,N,AD,LN,DATA,Q,ERRORA)

Arguments

F: CAMAC function code.

B. Branch number.

C: Crate number.

The crate number may be positive, negative, or zero. If it is positive and less than or equal to the number of crates per branch, a single crate operation is specified. Crate number = 0 indicates a branch controller function. If the crate number is between -1 and -127, then its absolute value, taken as a binary number, indicates which crates are to participate in a parallel operation. If the nth bit from the right is a "1," then crate n is addressed (e.g., C = -3 addresses crates 1 and 2). Crate number = -128 indicates all on-line crates are addressed in parallel.

N: Station number.

AD: Subaddress.

LN: Repeat count and word length specification. If F specifies a data transfer function (read or write), the absolute value of LN indicates the number of words to transmit; the sign indicates computer-length words if positive and CAMAC-length words if negative. If F specifies a control or test function, LN must be positive, and it indicates the number of times to repeat the operation.

DATA: Data block.

"DATA" must be an array of sufficient length to supply or to receive the data transmitted by the subroutine while executing the CAMAC command. "DATA" supplies data during write operations and receives data during read operations. If the computer word length is less than 24 bits and LN is negative for a read or write function, each CAMAC word will be stored in a contiguous block within the array "DATA" which will be the smallest integral number of words required to hold 24 bits. The word within the block having the greatest address is filled with the lowest order bits of the CAMAC word. Computer words having successively smaller addresses are filled with successively higher order bits of the CAMAC word until the last, which is filled out with high-order zeros if necessary. Note that although B,C,N, and AD remain constant in this subroutine, the computer storage address is incremented for each transmission.

Q: Q response from the last execution of the function.

ERRORA: Error vector.

Errors detected by the hardware or errors detected in arguments are indicated in this array. If the first word of the error vector is set to 0, i.e., ERRORA(1)=0, no errors were detected. If ERRORA(1)>0, then it

contains an error code, and the following three words may contain supplementary information about the error. The codes are:

1. Dataway timing error
2. Branch highway timing error (e.g., offline crate)
3. Illegal branch number
4. Illegal crate number (e.g., crate number greater than 7)
5. No X response
6. Illegal function code (e.g., not in the range 0 to 31)
7. Illegal station number (e.g., not in the range 0 to 31)
8. Illegal subaddress (e.g., not in the range 0 to 15)
9. Illegal value of LN
10. Starting CAMAC address is less than ending CAMAC address (see CMCASC)

ERRORA(2) contains the execution count at the time the error was detected. The subroutine stops execution when an error occurs and does not attempt to complete the requested number of function executions. ERRORA(3) is used as a cumulative error count. Whenever an error is detected, its value is increased by 1.

II. CMCSEQ

Purpose

To execute a single CAMAC function at a succession of CAMAC addresses.

Calling Sequence

CALL CMCSEQ(F,BA,CA,NA,ADA,LN,DATA,Q,ERRORA)

Arguments

F: CAMAC function code.
BA: Branch numbers.
CA: Crate numbers.
NA: Station numbers.
ADA: Subaddresses.
LN: Repeat count and word length specification. (+ = computer length; - = CAMAC length.)
DATA: Data block.
Q: Q response from last execution of function.
ERRORA: Error vector. (Integer array of length 4)

FORTRAN Definition (See page 82.)

```
      SUBROUTINE CMCSEQ(F,BA,CA,NA,ADA,LN,DATA,Q,ERRORA)
      COMMON/CMCCOM/NCOMP,NOMOD,NOCRT
      INTEGER F,BA(2),CA(2),NA(2),ADA(2),DATA(2),Q,ERRORA(4)
C
C     SET NUMBER OF WORDS PER TRANSFER.
C
      NWDS=NCOMP
      IF(LN.GT.0)NWDS=1
C
C       SET NUMBER OF EXECUTIONS.
C
      K=IABS(LN)
      IF(K.EQ.0) GO TO 30
C
C       LOOP THROUGH CALL TO SUBROUTINE CMCBSC.
C
      DO 10 I=1,K
      M=I*NWDS-NWDS+1
      CALL CMCBSC(F,BA(I),CA(I),NA(I),ADA(I),ISIGN(1,LN),DATA(M),Q,
     1   ERRORA)
C
C       CHECK FOR ERROR.
C
      IF(ERRORA(1).NE.0) GO TO 20
   10 CONTINUE
      RETURN
   20 ERRORA(2)=I
      RETURN
   30 ERRORA(1)=9
      ERRORA(2)=0
      ERRORA(3)=ERRORA(3)+1
      RETURN
      END
```

Note: Subroutine "CMCSEQ" could just as well serve as the fundamental CAMAC subroutine as subroutine "CMCBSC" since for LN=1 or LN=-1 they are identical in function, and any other sequence of CAMAC operations can be constructed from them.

Fig. 2

III. CMCASC

Purpose

To execute a specified CAMAC function in the address scan mode.

Calling Sequence

CALL CMCASC(F,BI,CI,NI,AI,BF,CF,NF,AF,LN,DATA, ERRORA,NEX)

Arguments

F: CAMAC function code.

BI: Initial branch number.

CI: Initial crate number.

NI: Initial station number.

AI: Initial subaddress.

BF: Final branch number.

CF: Final crate number.

NF: Final station number.

AF: Final subaddress.

LN: Execution limit and word size indication. (+ = computer length; - = CAMAC length.)

DATA: Dimensioned array for data sent or received.

ERRORA: Error vector.

NEX: Number of times function was executed with a Q=1 response, (number of words transmitted if function is read or write).

Addressing Rule

The specified function is executed first at the address given by BI,CI,NI,AI. Then if the Q response is 1, the subaddress is incremented by 1 and the index into the data block incremented by either 1 or the number of computer words necessary to contain a CAMAC word, depending on the sign of LN, and the function executed at this new subaddress. If the Q response is 0, the subaddress is set to zero, the data block index is not changed, and the station number is incremented by 1 for the next execution. When the station number is incremented beyond the value of "NOMOD," it is set back to 1 and the crate number is incremented. If the crate number is incremented beyond the value of "NOCRT," it is set back to 1, and the branch number is incremented. The total number of executions of the function is limited by the magnitude of LN and by the stated final values of the CAMAC address, BF,CF,NF,AF. Special attention must be paid to the X response in the address scan mode. When a module responds with Q=0, it may, at the same time, give X=1 or X=0; consequently, in this mode of access, the X response is ignored when the Q response is 0.

FORTRAN Definition (See Fig. 3.)

IV. CMCRPT

Purpose

To execute a specified CAMAC function in the repeat mode.

Calling Sequence

CALL CMCRPT(F,B,C,N,AD,LN,DATA,ERRORA)

Arguments

F: CAMAC function code.

B: Branch number.

C: Crate number.

N: Station number.

Ad: Subaddress.

LN: Execution limit and word size indication. (+ = computer length; - = CAMAC length.)

DATA: Dimensions array for data sent or received.

ERRORA: Error vector.

Execution Rule

In the repeat mode, the CAMAC address (B,C, N,A) is never changed, but the single address is expected to supply many words of data. Q is used as a timing signal; Q=1 indicates that the previously executed function succeeded; Q=0 indicates that the module was not ready to accept the function and that the controller should try again.

FORTRAN Definition (See Fig. 4.)

V. CMCSTP

Purpose

To execute a specified CAMAC function in the stop mode.

Calling Sequence

CALL CMCSTP(F,B,C,N,AD,LN,DATA,ERRORA,NEX)

Arguments

F: CAMAC function code.

B: Branch number.

C: Crate number.

N: Station number.

Ad: Subaddress.

LN: Execution limit and word size indication. (+ = computer length; - = CAMAC length.)

DATA: Dimensioned array for data sent or received.

ERRORA: Error vector.

NEX: Number of times function was executed.

Execution Rule

In the stop mode, the CAMAC address (B,C,N,A) is never changed, but the single address is expected to supply (or accept) many words of data. It is

```
      SUBROUTINE CMCASC(F,BI,CI,NI,AI,BF,CF,NF,AF,LN,DATA,ERRORA,NEX)
      COMMON/CMCCOM/NCOMP,NOMOD,NOCRT
      INTEGER F,BI,CI,AI,BF,CF,AF,DATA(2),ERRORA(4)
      INTEGER B,C,A,Q
C
C       SET MAXIMUM NUMBER OF EXECUTIONS.
C
      K=IABS(LN)
C
C       SET NUMBER OF WORDS PER DATA TRANSMISSION.
C
      NWDS=NCOMP
      IF(LN.GT.0) NWDS=1
C
C       CHECK FOR ARGUMENT ERROR.
C
      IF(K.EQ.0) GO TO 50
      IF(BF.GT.BI) GO TO 5
      IF(BF.LT.BI) GO TO 60
      IF(CF.GT.CI) GO TO 5
      IF(CF.LT.CI) GO TO 60
      IF(NF.GT.NI) GO TO 5
      IF(NF.LT.NI) GO TO 60
      IF(AF.LT.NI) GO TO 60
C
C       SET INITIAL VALUES OF B, C, N, A.
C
    5 B=BI
      C=CI
      N=NI
      A=AI
C
C       J IS USED TO INDEX THE ARRAY "DATA".
C       I COUNTS THE NUMBER OF EXECUTIONS OF THE FUNCTION.
C
      J=1
      I=0
C
C       HAVE WE EXCEEDED THE MAXIMUM ALLOWABLE NUMBER OF EXECUTIONS OF
C             THIS FUNCTION?
C
   10 IF(I.GE.K) GO TO 30
C
C       EXECUTE THE FUNCTION ONCE.
C
      CALL CMCBSC(F,B,C,N,A,ISIGN(1,LN),DATA(J),Q,ERRORA)
C
C       CHECK FOR ERROR.
C
      IF(ERRORA(1).NE.0) GO TO 28
C
C       CHECK Q.
C
      IF(Q.EQ.0) GO TO 40
C
C       INCREMENT THE CORE ADDRESS INDEX AND THE CAMAC SUBADDRESS.
C
      I=I+1
      J=J+NWDS
      A=A+1
      IF(A.GT.15) GO TO 40
```

Fig. 3
(Page 1 of 2)

```
C
C     CHECK FOR END OF RANGE.
C
   20 IF(B.LT.BF) GO TO 10
      IF(B.GT.BF) GO TO 30
      IF(C.LT.CF) GO TO 10
      IF(C.GT.CF) GO TO 30
      IF(N.LT.NF) GO TO 10
      IF(N.GT.NF) GO TO 30
      IF(A.LE.AF) GO TO 10
      GO TO 30
C
C     B,C,N,A, OR I HAS EXCEEDED LIMIT; HALT EXECUTION OF
C       CAMAC FUNCTION.
C
   28 IF((ERRORA(1).EQ.5).AND.(Q.EQ.0))GO TO 38
   30 NEX=I
      RETURN
C
C     GO TO NEXT MODULE
C
   38 ERRORA(1)=0
      ERRORA(3)=ERRORA(3)-1
   40 A=0
      N=N+1
      IF(N.LE.NOMOD) GO TO 20
C
C     GO TO NEXT CRATE.
C
      N=1
      C=C+1
      IF(C.LE.NOCRT) GO TO 20
C
C     GO TO NEXT BRANCH.
C
      C=1
      B=B+1
      GO TO 20
C
C     ERRORS DETECTED IN ARGUMENTS.
C        1. LN=0
C
   50 ERRORA(1)=9
      ERRORA(2)=0
      ERRORA(3)=ERRORA(3)+1
      RETURN
C
C     2. (BI,CI,NI,AI) IS GREATER THAN(BF,CF,NF,AF)
C
   60 ERRORA(1)=10
      RETURN
      END
```

Fig. 3
(Page 2 of 2)

```
      SUBROUTINE CMCRPT(F,B,C,N,AD,LN,DATA,ERRORA)
      COMMON/CMCCOM/NCOMP,NOMOD,NOCRT
      INTEGER F,B,C,AD,DATA(2),ERRORA(4),Q
C
C     SET NO. OF COMPUTER WORDS PER TRANSFER.
C
      NWDS=NCOMP
      IF(LN.GT.0) NWDS=1
C
C     SET NUMBER OF EXECUTIONS.
C
      K=IABS(LN)
C
C      CHECK FOR ZERO.
C
      IF(LN.EQ.0)GO TO 40
C
C     SET EXECUTION PARAMETER FOR "CALL CMCBSC"
C
      L=ISIGN(1,LN)
C
C     INITIALIZE LOOP.
C
      J=1
      I=0
   10 IF(I.GE.K) RETURN
C
C     INCREMENT TALLY
C
      I=I+1
C
C     ATTEMPT TO EXECUTE FUNCTION.
C
   20 CALL CMCBSC(F,B,C,N,AD,L,DATA(J),Q,ERRORA)
      IF(ERRORA(1).NE.0) GO TO 30
      IF(Q.EQ.0) GO TO 20
C
C     INCREMENT STORAGE ADDRESS AND LOOP BACK.
C
      J=J+NWDS
      GO TO 10
C
C     ERROR.  STORE EXECUTION COUNT IN ERRORA(2).
C
   30 ERRORA(2)=I
      RETURN
C
C     LN=0. RETURN ERROR 9.
C
   40 ERRORA(1)=9
      ERRORA(2)=0
      ERRORA(3)=ERRORA(3)+1
      RETURN
      END
```

Fig. 4

assumed able to supply or accept a word of data whenever the controller addresses it, until the block is exhausted. The controller may terminate the process if the number of executions exceeds the limit given by the magnitude of LN, or the module may terminate the process by responding with Q=0. Q=0 implies that no data was sent or accepted during the CAMAC cycle for which it is the response. Q=1 indicates normal execution of the function.

FORTRAN Definition (See Fig. 5.)

VI. CMCLUP

Purpose

To execute a specified CAMAC function at a hierarchical sequence of addresses with the option of skipping certain portions of the sequence based on the Q response.

Calling Sequence

CALL CMCLUP(F,BA,LB,CA,LC,NA,LKN,ADA,LAD,LN, DATA,Q,QON,ERRORA,NEX)

```
      SUBROUTINE CMCSTP(F,B,C,N,AD,LN,DATA,ERRORA,NEX)
      COMMON/CMCCOM/NCOMP,NOMOD,NOCRT
      INTEGER F,B,C,AD,DATA(2),ERRORA(4),Q
C
C     SET NO. OF COMPUTER WORDS PER TRANSFER
C
      NWDS=NCOMP
      IF(LN.GT.0) NWDS=1
C
C     SET NO OF EXECUTIONS.
C
      K=IABS(LN)
      IF(K.EQ.0)GO TO 30
C
C     SET EXECUTION PARAMETER FOR "CALL CMCBSC"
C
      L=ISIGN(1,LN)
C
C     INITIALIZE LOOP.
C
      J=1
      I=0
C
C       TEST FOR END OF EXECUTION.
C
   10 IF(I.GE.K) RETURN
C
C     INCREMENT EXECUTION TALLY.
C
      I=I+1
C
C     ATTEMPT TO EXECUTE FUNCTION.
C
      CALL CMCBSC(F,B,C,N,AD,L,DATA(J),Q,ERRORA)
      IF(ERRORA(1).NE.0) GO TO 20
      IF(Q.EQ.0) GO TO 20
      J=J+NWDS
      GO TO 10
C
C     ERROR. SET EXECUTION COUNT IN NEX AND EXIT.
C
   20 NEX=I
      RETURN
C
C     ERROR: LN = 0.
C
   30 ERRORA(1)=9
      ERRORA(2)=0
      ERRORA(3)=ERRORA(3)+1
      GO TO 20
      END
```

Fig. 5

Arguments

F: CAMAC function code.

BA: Branch number array.

LB: Number of branch numbers.

CA: Crate number array.

LC: Number of crate numbers.

NA: Module number array.

LKN: Number of module numbers.

ADA: Subaddress array.

LAD: Number of subaddresses.

LN: Execution limit and word size indication. (+ = computer length; - = CAMAC length.)

DATA: Data array.

Q: Q response of last execution.

QON: Control argument indicating whether or not to use Q response for skipping addresses. 1 means use Q; Q means ignore Q.

ERRORA: Error vector.

NEX: Number of times the function was executed, or the number of times the function was executed with a Q=1 response if QON=1, (number of words transmitted for read or write functions).

Execution Rule

"BA," "CA," "NA," and "ADA," are each one-dimensional arrays containing lists of branch, crate, module, and subaddress numbers respectively. The program executes, or attempts to execute, the function at each CAMAC address which can be formed by combining the elements of these arrays. It begins by choosing the first element of each array and advances the address by choosing the next element of the subaddress array, etc., scanning the subaddresses most frequently, the module numbers next, etc., until the last element of each array is used. If QON=1, a Q=0 response will cause the module index to advance and the subaddress index to be set to 1.

FORTRAN definition (See Fig. 6.)

VII. REFERENCES

1. "CAMAC: A Modular Instrumentation System for Data Handling," TID-25875.

2. "CAMAC: Organization of Multi-Crate System," TID-25876.

3. "Supplement to CAMAC Instrumentation System Specifications," TID-25877.

```
      SUBROUTINE CMCLUP(F,BA,LB,CA,LC,NA,LKN,ADA,LAD,LN,DATA,Q,QON,
     1    ERRORA,NEX)
      COMMON/CMCCOM/NCOMP,NOMOD,NOCRT
      INTEGER F,BA(LB),CA(LC),NA(LKN),ADA(LAD),DATA(2),Q,QON,ERRORA(4)
C
C     SET NUMBER OF COMPUTER WORDS PER TRANSFER.
C
      NWDS=NCOMP
      IF(LN.GT.Ø) NWDS=1
C
C     SET EXECUTION LIMIT.
C
      K=IABS(LN)
C
C     CHECK LIMIT FOR ZERO.
C
      IF(K.EQ.Ø) GO TO 6Ø
C
C     SET EXECUTION PARAMETER FOR "CALL CMCBSC".
C
      L=ISIGN(1,LN)
C
C     INITIALIZE INDEXES FOR EXECUTION COUNT AND DATA TRANSFER.  I
C         COUNTS EXECUTION OF F; J INDEXES "DATA".
C
      I=Ø
      J=1
```

Fig. 6
(Page 1 of 2)

```
C
C     SET UP FOUR NESTED DO'S TO EXECUTE THE ADDRESSING ALGORITHM.
C
      DO 40 MB=1,LB
      DO 40 MC=1,LC
      DO 40 MN=1,LKN
      DO 30 MAD=1,LAD
C
C     ATTEMPT TO EXECUTE FUNCTION ONE TIME.
C
      CALL CMCBSC(F,BA(MB),CA(MC),NA(MN),ADA(MAD),L,DATA(J),Q,ERRORA)
C
C     IS Q BEING CHECKED?
C
      IF(QON.EQ.0) GO TO 20
C
C     YES.  IF Q=1,PROCEED NORMALLY.  IF Q=0, IGNORE X RESPONSE AND
C         BRANCH TO END OF N LOOP.
C
      IF(Q.EQ.1) GO TO 20
      IF((ERRORA(1).EQ.5).OR.(ERRORA(1).EQ.0)) GO TO 40
      GO TO 70
C
C     INCREMENT DATA INDEX AND EXECUTION COUNT.  TEST EXECUTION LIMIT.
C
   20 I=I+1
      J=J+NWDS
      IF(I.GE.K) GO TO 50
C
C     DO NORMAL ERROR CHECK.
C
      IF(ERRORA(1).NE.0) GO TO 70
C
C     END OF SUBADDRESS LOOP.
C
   30 CONTINUE
C
C     END OF BRANCH, CRATE, MODULE LOOPS.
C
   40 CONTINUE
C
C     SET EXECUTION COUNT AND RETURN
C
   50 NEX=I
      RETURN
C
C     LN=0. GIVE TYPE 9 ERROR.
C
   60 ERRORA(1)=9
      ERRORA(2)=0
      ERRORA(3)=ERRORA(3)+1
      RETURN
C
C     ERROR.  SET ERRORA(2) TO EXECUTION COUNT.
C
   70 ERRORA(2)=I
      GO TO 50
      END
```

Fig. 6
(Page 2 of 2)

PRECEDING PAGE BLANK-NOT FILMED

STANDARD SOFTWARE FOR CAMAC - COMPUTER SYSTEMS

by

I. N. Hooton,

Chairman of the ESONE Software Working Group,

A.E.R.E., Harwell, England.

Abstract

The features of the CAMAC Language of interest to the application programmer are briefly described. A method of implementation via the CAMAC intermediate language, IML, is discussed.

Appendices give examples of statements in the CAMAC Language and of translations of typical statements into IML.

Contents

1. Introduction.

2. Background.

3. Features of the CAMAC Language.

4. The concept of the Virtual Controller.

5. Implementation.

6. Acknowledgements.

Appendix 1. Examples of CAMAC Language Statements.

Appendix 2. Examples of CAMAC to IML Translation.

ESONE SWG 17/72

October 1972.

1. Introduction

The Software Working Group of the ESONE Committee has published a "Proposal for a CAMAC Language" in the form of a language specification. The proposal includes an appendix with two extensive examples of the use of the language, but is not otherwise directed specifically to the application programmer.

In this paper the background is discussed in order to show the need for an agreed language. This is followed by a description of some of the facilities available to the application programmer and the final section discusses a method of implementation. Examples of specific statements in the language are given in Appendix 1. The translation of typical statements into a more explicit language is discussed in Appendix 2.

Since the "Proposal for a CAMAC Language" has been published expressly to obtain comments the final version may differ from that described here.

2. Background

A register in CAMAC hardware is located at a subaddress (A) of a station (N) in a crate (C) on a branch (B). Its CAMAC address is the combination of the numerical values for each of the four components B, C, N and A. The action to be performed is specified by the CAMAC function code (F) directed to the CAMAC address. Every action generates a Command Accepted (X) signal from CAMAC and may also generate a Response (Q) signal. Certain functions also cause up to 24-bits of data to be read from or written to CAMAC.

Because of the amount of information required for a single CAMAC operation the hardware which couples CAMAC to the computer contains a number of registers. The distribution of the information between these registers is usually arranged to suit the hardware of the computer. For example, in one commercially available coupler the crate address (C) and the function code (F) are combined and held in one register. Another register holds the high-order data bits (17 to 24), and B, N and A form part of a MOVE instruction applied to the low-order data bits (1-16). This gives efficient use of the computer hardware, particularly with a sequence of instructions which address the same action to different registers in a common crate.

One method of programming CAMAC operations for such a coupler is to write specific instructions within the application program which load the appropriate registers. This gives the fastest performance in execution for two reasons. Firstly, the overheads are minimised and secondly, only those features required are used for each operation. For example, if no action is to be taken on the state of Q, the Q Response is not tested.

The major disadvantage of this method is that it requires the application programmer to have detailed knowledge of the coupler. He must know, for example, the different sequences for Read and Write operations and the effect of 16-bit or 24-bit data transfers on them. He must know how to access Q and X, and in particular he must convert the CAMAC BCNA address into appropriate forms. In short, he is coding for a very specific machine.

This difficulty may be overcome by providing either a set of macro calls which generate the appropriate in-line code or a set of subroutines as a driver package for the coupler. In either case the set may be implemented by an expert in the computer-coupler combination and then made available to application programmers.

Effectively the macro or subroutine call defines what is to happen in the CAMAC part of the total system and is independent of the mechanism in the coupler. However, the call must define explicitly the mechanism within CAMAC. For example, the demand signal in a Crate Controller Type A is enabled by means of CAMAC function code (26) to subaddress (10) of station (30) in the crate (e.g. crate 2, branch 1), and a suitable instruction would be

```
CTRL 26 H,1,2,30,10
```

An alternative way of defining precisely the same requirement is

```
ENABLE CRATEDEMAND
```

The first example is from a fully explicit language, designed for ease of implementation (for example by macros), and the second is from a language designed for ease of understanding by the programmer. Thus there is a need for two different languages having the same facilities. The ESONE Software Working Group recognises this. The second statement is from the CAMAC Language and the first is from a CAMAC intermediate language, IML. The aim of the CAMAC

language is to enable the programmer to specify his requirement for each operation rather than the detailed mechanism involved. IML is designed to be totally explicit, that is each statement contains within itself all the information required for its execution. Both languages have the same facilities, though IML may require more statements to express them.

3. Features of the CAMAC Language

In the definition of the CAMAC Language the run time features are expressed in transfer, control, branching and executive statements. Transfer statements deal with reading data from or writing data to CAMAC. Each statement specifies the action to be performed, the CAMAC address and a reference to the computer memory. Control statements do not include the memory reference and may be addressed either to a CAMAC address, for example to clear a register, or to the coupler, for example to enable demands. Branching statements may be unconditional, conditional on the state of the LAM or status at a CAMAC address, or conditional on flags in the coupler. Executive statements relate primarily to the scheduling of tasks.

The actions to be performed in the CAMAC hardware may be defined by the CAMAC function code, e.g. F(0); by a standard mnemonic, e.g. READ; or by a user defined symbol, e.g. COPY.

The CAMAC address may be contained within the statement or held in a computer location pointed to by the statement. This latter form permits run-time modification of the CAMAC address. The address may be written in the BCNA form or as a name declared by the user. When repeated use is made of the same group of CAMAC registers it is useful to declare a name for the group. The form of declaration shows whether the group has a random distribution of addresses (described by a list) or whether it is uniform and may be accessed by incrementing the components of a CAMAC address (for example by the Address Scan mode of operation).

The reference to computer memory may be to a variable, an array or a list. Each element may contain a computer length word (typically 16-bits) or a CAMAC length word (24-bits). A list is a specialised form of one dimensioned array in which successive accesses are directed to consecutive elements of the list. This form is directly applicable to loading a buffer with information available at irregular intervals.

Write statements may contain the actual value (literal) that is to be transferred to CAMAC.

Transfer and control statements may include a repeat qualifier giving the number of times the operation is to be performed before executing the next statement. This enables block transfers to be specified. The declaration statements allow the standard forms of Q response to be associated with CAMAC addresses and then used in transfer and control statements.

Demand handling is simplified by the language. Statements are provided for specifying the various mechanisms available in CAMAC and for linking demands with labelled statements or tasks.

Appendix 1 gives some examples of the types of statement that may be made in the CAMAC Language.

4. The Concept of the Virtual Controller

The specification of the language implies that certain facilities are available in the controller, and can therefore be regarded also as the specification of an ideal (or virtual) controller. Different implementations of this virtual controller may use different combinations of hardware and software; for example, an Address Scan may be implemented in one system by automatic hardware and in another by a program loop. The times taken will differ considerably but both systems interpret the same statement to produce the same end result. It is this concept of the virtual controller which makes both IML and the CAMAC Language implementation independent.

5. Implementation

The CAMAC Language is designed primarily for easy programming. It has powerful hardware declarations, simple action statements and a free format. The intermediate language, IML, is designed primarily for easy implementation. It has elementary declarations, detailed action statements and a rigid format. Both languages are implementation independent and have been designed expressly for CAMAC-computer systems.

The translation of the CAMAC Language may therefore be performed in two sections. Firstly, CAMAC is translated into IML. Since both the input and the output are implementation independent all

users could, in principle, share the same translator. Secondly, IML is translated into a form acceptable by the local computing system.

This method has a major advantage when the facilities required increase with time. A minimum set of statements in IML may be selected for implementation initially in a local translator. When the set is expanded it automatically brings the capability of exploiting the corresponding CAMAC Language statements provided by a CAMAC to IML translator.

The designer of specific computer-coupler combinations has a standard interface to the CAMAC hardware (the CAMAC Branch Highway port). The definition of the facilities in IML and the CAMAC Language (the specification of the virtual controller) provides a standard interface to application programs.

6. Acknowledgements

This report has not been reviewed by the Software Working Group of the ESONE Committee and must therefore be taken as the personal views of the author. Nevertheless it could not havebeen produced without the collaboration and advice of the members of the working group. Were it not for the enthusiasm and hard work of the group and of our colleagues in the U.S.AEC NIM-CAMAC Software Working Group the two languages discussed would not exist.

Appendix 1.

Examples of CAMAC Language Statements.

A1.1 Hardware Declaration Statements

Declaration statements allow the detailed addressing of the CAMAC hardware to be expressed in an initial section of the program and then referenced by user-defined names. The names declared here will be used in the later examples and in Appendix 2.

MYCRATE = B(1)C(2)

This assigns the name MYCRATE to crate 2 on branch 1. The name may be used in subsequent declarations as part of a hierarchic technique for generating addresses. If the crate is relocated a modification to this one declaration will amend all the derived addresses.

REG = MYCRATE N(1)A(0)

The register REG is at subaddress 0 of station 1 in MYCRATE.

DEVICES (1:4) = MYCRATE N(3, 7, 2, 5)A(0)

The registers at subaddress 0 of stations 3, 7, 2 and 5 in MYCRATE form the first, second, third and fourth elements respectively in the hardware array DEVICES.

COUNTERS (1:16) = MYCRATE N(10:16:2)A(0:3)Q

The registers at subaddresses 0, 1, 2 and 3 at each of the stations from 10 to 16 by 2 (10, 12, 14 and 16) in MYCRATE form the first sixteen elements of the hardware array COUNTERS. The qualifier Q indicates that the array is to be accessed in Address Scan (Q Scan) mode.

BUFFER = MYCRATE N(18)A(0) S

The register BUFFER is at subaddress 0 of station 18 in MYCRATE. The qualifier S indicates that BUFFER is to be accessed with the Stop Mode of Q; that is, it generates Q = 1 as long as words from a data block are being transferred.

A1.2 Action Statements

READ REG LOCN

or F(0) B(1)C(2)N(1)A(0) LOCN

Both the above statements mean 'read the contents of the register (named REG) at subaddress 0 of station 1 in crate 2 on branch 1 into the computer memory location LOCN'.

WRITE LOCN REG

Write the contents of memory location LOCN into register REG.

CLEAR REG

Reset all the bits of register REG to zero.

IF LAM REG GOTO LABEL

If the register REG is generating a look-at-me signal branch to the statement labelled LABEL.

IF Q GOTO LABEL

If the Q flag in the coupler is set (as a result of the previous CAMAC operation) branch to the statement labelled LABEL.

ENABLEINT MYCRATE

Permit demand signals in MYCRATE to generate demands to the computer.

CLEAR DEVICES (1:4)

Reset to 0 all the bits of the registers constituting elements one to four of the array DEVICES.

READ COUNTERS (1:16) ARRAY (1:16)

Read the contents of the registers constituting the first sixteen elements of the hardware array COUNTERS into the corresponding elements of the software array ARRAY. Since COUNTERS was declared with the qualifier Q use the Address Scan mode of access.

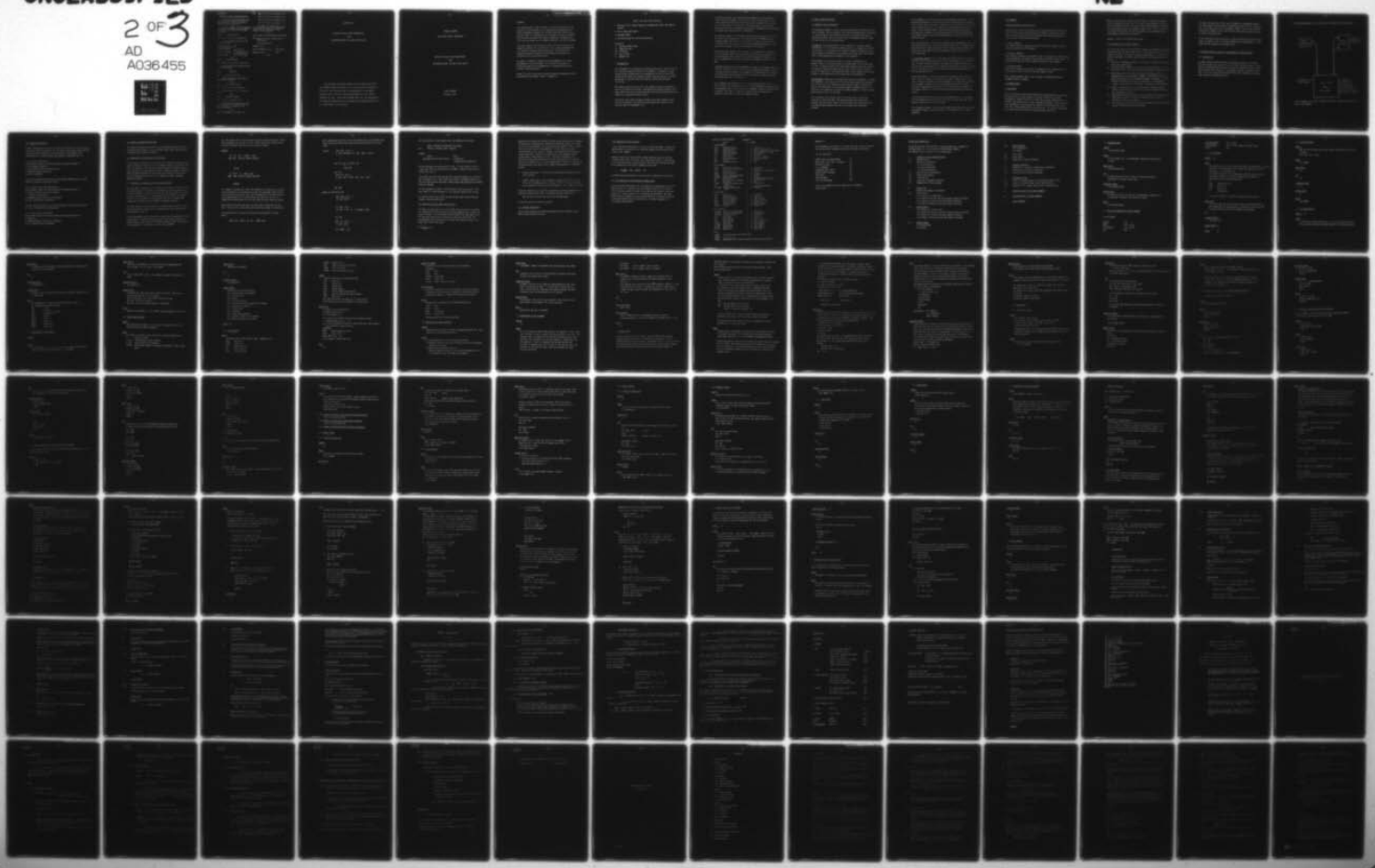
REPEAT (64) READ BUFFER LIST LISTFULL

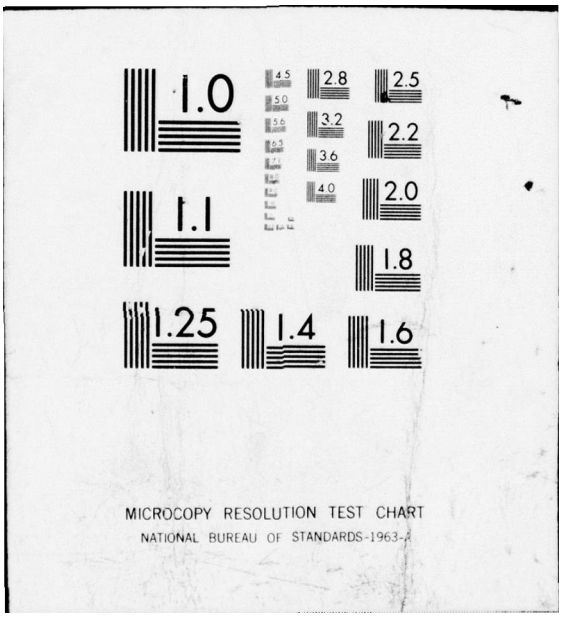
Read the contents of the register BUFFER into the next element of the software list LIST. Repeat this operation 64 times; or until the Response Q = 0 is obtained; or until the list is full, in which case branch to the statement labelled LISTFULL.

AD-A036 455 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
A036455




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Appendix 2.

Examples of CAMAC to IML Translation

Since both the CAMAC Language and the intermediate language IML have still to be revised the final versions may differ from those shown here.

In these examples the CAMAC Language statements are similar to those explained in Appendix 1 and make use of the same declarations.

CAMAC: READ REG LOCN

IML: READ 0 H,1,2,1,0 V,16,LOCN N,N,N

The IML statement consists of the following fields:-

Statement class	READ
CAMAC Function Code	0
Hardware Address	H, 1, 2, 1, 0 (B, C, N, A)
Software Reference	V, 16, LOCN (a variable of 16-bits)
Condition Code	N, N, N (N = null)

CAMAC: WRITE LOCN REG

IML: WRITE 16 H,1,2,1,0 V,16,LOCN N,N,N

Write statements have similar fields to read statements.

CAMAC: CLEAR REG

IML: CTRL 9 H,1,2,1,0 N,N

Control statements do not require the software reference.

CAMAC: IF LAM REG GOTO LABEL

IML: CTRL 8 H,1,2,1,0 N,JQ,LABEL,N

The condition is 'If Q = 1 jump to LABEL'.

CAMAC: ENABLEINT MYCRATE

IML: CTRL 26 H,1,2,30,10 N,N,N

The IML statement uses function code 26 at subaddress 10 of station 30 in the crate MYCRATE as specified for Crate Controllers Type A.

CAMAC: READ DEVICES (1:4) ARRAY (1:4)

IML: READ 0 H,1,2,3,0 A,2

READ 0 H,1,2,7,0 A,2

READ 0 H,1,2,2,0 A,2

READ 0 H,1,2,5,0 A,2

IML expands the CAMAC four statements inserting t values from the hardware an arrays into each statement.

CAMAC: REPEAT (64) READ BUF

IML: READ 0 H,1,2,18,0 L,16, LISTFULL,R,64

The conditions are:

transfer if Q = 1

jump to LISTFULL if Q = 0

repeat 64 times

SECTION IV

REPORT ON REAL-TIME EXTENSIONS
AND
IMPLEMENTATIONS OF REAL-TIME BASIC

The European Regional Branch of the Industrial Real-Time BASIC Committee made as its first order of business a review of the existing sets of extensions to the BASIC language now in use or under development by the various vendors and use. This new document has not yet appeared in the Workshop Minutes but is produced here anyway because of its importance to the field.

PURDUE EUROPE

REAL-TIME BASIC COMMITTEE

REPORT ON REAL-TIME EXTENSIONS AND IMPLEMENTATIONS OF REAL-TIME BASIC

FIRST REPORT

October 1975

PRECEDING PAGE BLANK-NOT FILMED

Foreword:

The increasing use of small computers in real-time applications has resulted in the requirement for an appropriate programming system. Simple programming languages such as BASIC have been widely used to meet this need. The language has been extended to include statements handling real-time events and process peripherals. Implementations utilising such extensions are commonly referred to as real-time BASIC.

This paper compares and discusses various real-time extensions and suggests guidelines for a unification. It is a working document and is intended to stimulate further discussion. In particular the examples used present semantic interpretation and should not be considered to be a syntax definition.

The paper is intended to inform users and implementors of current developments in real-time BASIC. It is the aim of the group to produce more definitive recommendations in conjunction with users and other interested bodies.

Comments from any source would be most welcome by the group and should be addressed to any member listed in appendix V.

BASIC FOR REAL-TIME SYSTEMS

1. Objectives for a simple language to program small real-time computer systems.
2. How to reach these goals.
3. Dartmouth BASIC.
4. Functional needs for real-time extensions.

Appendices:

I. Dartmouth BASIC syntax.
II. Comparison part.
III. Literature
IV. User's guide ...
V. Address list

1. Introduction:

The achievements of semiconductor technology have made it possible to use mini and micro computer systems for RT problems. In contrast to large process-control systems which run autonomously and restrict user interactions we are faced with computer systems, that are used as a tool by a technician or scientist requiring intensive interaction and great flexibility.

Some typical application areas for such systems are test and diagnosis, control of industrial processes, experiments and teaching systems. All of these require the user to perform multiple test runs and he must have the capability to make modifications easily.

Flexibility often means increased software costs while hardware costs continue to decrease. The software system become more complex and the training costs for programmers rise.

In addition to that, even experienced programmers need a great amount of time to be able to program technical and scientific applications. This causes delay and difficulties and hinders quick solutions. This problem only can be solved by the user writing programs himself. But for this purpose he must have a programming system with a language easy to learn and to remember and producing results immediately.

The computer system as a tool should provide strong support for the user. The level of the language should be high (similar to FORTRAN) and the language elements should allow application to a large class of problems. The syntax should be simple and should allow immediate diagnostics. There should not be complex relationships between the statements.

Working with the computer should be done in a dialogue form. The system control commands (such as saving programs) and the editing commands (such as deleting) should be powerful but as few as possible. Furthermore it must be simple and fast to pass from editing a program to running it and back.

Immediate execution of single statements or groups of statements should be possible. A tracing facility should exist for debugging purposes. Values of variables should be readable and changable. The user should not be forced to call and start system programs such as an editor or loader but rather the system should be a unified whole.

Let's summarize the objectives such a system should be able to meet. It should allow quick, portable solutions of simple problems for beginners and non-programmers and it should be easy to implement in order to achieve wide acceptance. The latter is especially important for the education of newcomers to the field of real-time computer control.

2. How to reach these goals

2.1 Compiler versus Interpreter

A programming language is used to describe and specify the program flow to solve a problem. But the language itself must be implemented by translating it into the code of the host machine. We try to describe three ways of implementing BASIC. There are hybrid forms in existence but most of them are near to one of the following types.

A) Compiler: The processor which "runs a program" is executing machine instructions, i.e. at runtime the program exists as a sequence of machine instructions. It is the task of the compiler program to check the syntax and to translate the BASIC source program into machine code. This machine code is then executed by the hardware of the processor.

Because BASIC is not block oriented it is easy to implement with a compiler. Most of the syntax checks can be done when the lines are typed in. This is important when using intelligent terminals where the terminal can make the line-syntax check and only send correct lines to the main system for compilation. The compilation time is thus reduced. But for any changes in the program the entire program has to be recompiled.

B) Interpreter: BASIC can be implemented by using an interpreter, a program that examines, at run-time, each line of the source program starting with the first and calls the appropriate machine-code routine to perform the action. The interpreter in a way executes BASIC statements directly, i.e. looking at the system one can speak of a BASIC computer. In fact a virtual BASIC machine is built by software.

The translation of the source code at run time causes a significant reduction of execution speed. The program coded in the higher-level-language code (the source code) is more core efficient than machine code but machine-coded programs are able to run without the interpreter routines.

Micro programs are an interesting form of interpretive execution. In this approach the processor is executing primitive instructions at high speed from a special high speed memory. Sequences of such instructions are called by the instructions of the main memory which are generated by the interpreter. If the highspeed memory contains routines which execute BASIC statements as a sequence of primitive instructions one has implemented a direct BASIC source code executing machine which is much faster than the software version.

Most new interpreters translate the statements into "intermediate code" to improve execution speed. This means that the source code is changed into a form which allows faster execution but still holds all the information to reclaim the source code for listing purposes. The user thus deals only with source code.

C) Incremental compiler: The line orientation of BASIC makes it possible to compile it line by line without much loss of efficiency. (This would not be true for block oriented languages like ALGOL.) The program is translated line by line into machine code and stored by placing the starting locations into a list. When modifications or deletions of lines are performed this is done by changing the list.

Before the program is run one must check the statements which influence other lines i.e. nesting of FOR NEXT loops, GOTO's, GOSUB's etc.). This check is usually triggered by the RUN statement. It is possible to design the incremental compiler such that the emitted code with its forward reference linkage and starting address list can be executed without run-time support making its run-time efficiency close to that of an ordinary compiler.

A difficulty arises from the fact that the sourcecode can't be reclaimed from the compiled code so that one has to keep a source code copy of the program in addition to the compiled code.

The incremental compiler usually needs background storage but by combining advantages of compiler and interpreter it is a powerful type of implementation.

2.2 Commands

Extensions towards operating system

When writing, testing and running programs the system should support the user. For this purpose system commands are added so that the user gets familiar with the language and the system at the same time and need not learn several different system programs.

A) Editor commands
All commands which are needed for editing like insert, delete, list etc. should be contained in the language.

B) Execution commands
The starting or stoping of a program or part of a program, the execution of single statements, the display of calculated or defined variables after the program has halted and similar commands should be in the language.

C) Storage commands
These include storage or retrieval of programs or data, defined input of data blocks or text or calls for the next segment of code.

Most of these commands should also be usable in immediate mode and as statements within a program.

3. Dartmouth BASIC

3.1 Why BASIC?

Most of the user-oriented and problem-oriented languages were designed for the special use of certain groups of experts. Hence, most of these languages are somewhat difficult to learn and to apply. Recognizing this problem it was decided that a simple language was needed. It must have very simple grammatical rules and must be capable of being learned in a very short time. Thus Dartmouth BASIC came into being. It is easy to learn and can be applied to most computing problems quickly and easily.

BASIC also does have the capability to be used in interactive programs, that is, programs that require or allow user's participation in order to achieve the desired results. Because of these features BASIC has become a de facto standard on most minicomputer systems. This is still true though it has some shortcomings and therefore should not be used where these shortcomings cause serious restrictions.

Appendix I shows the Dartmouth BASIC syntax.

3.2 Limitations of the BASIC languages

Since the time BASIC was designed several changes in programming philosophy have taken place. The modern approach is to write GOTO-less well structured programs. In order to understand a program dynamically it is important to minimise the number of possible execution paths. Shortcomings and consequences:

a) BASIC has no COMPOUND statement. This results in additional GOTO's and makes the program opaque.

b) The CONDITIONAL statement is primitive by only allowing a line number behind the 'THEN'. This again increases the number of execution paths.

c) BASIC does not provide a general subroutine mechanism. The only characteristics of a subroutine in BASIC is the RETURN to the line after the calling line by the RETURN statement. In general it is not possible to indicate the body of a subroutine. It is possible to scatter a subroutine over the program. (Certainly not recommended.)

d) BASIC is not appropriate for writing symbolic programs because symbolic labels are not allowed. This fact also causes difficulties for rearranging a program.

e) BASIC does not use the concept of local variables. All variables are global and restricted to one or two characters thus making it difficult to write long programs.

f) The limitation of data types to real and string restricts the possibilities of defining other data structures.

All these items may restrict the use of RT-BASIC for programming complex real-time systems. Problems where disadvantages caused by these limitations can't be avoided should not be programmed in BASIC. These are not inherent limitations of BASIC systems. Some of these limitations have been overcome by certain implementations.

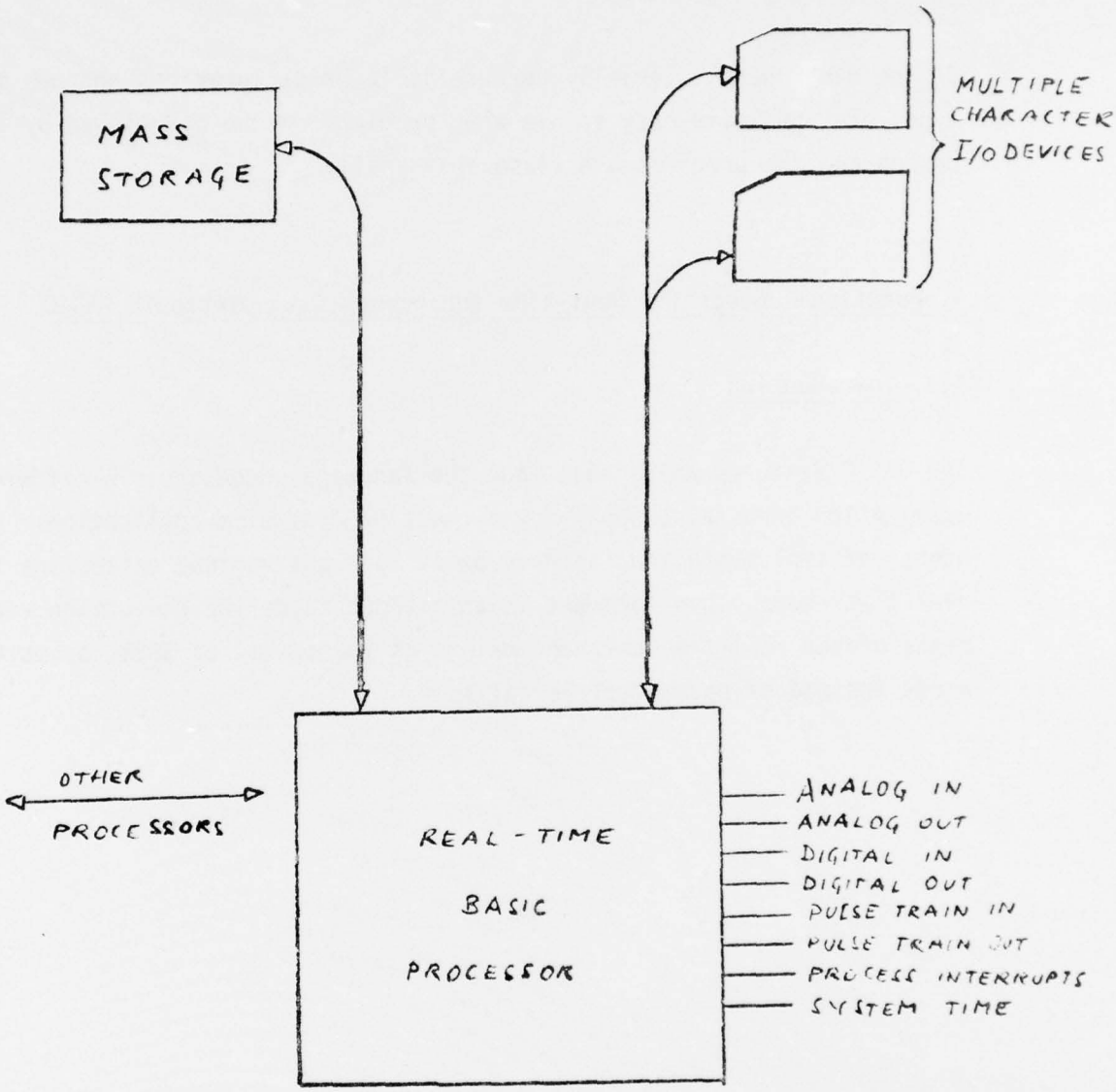
On the other hand especially because it lacks declarations and has few data types etc. BASIC is easy to use when problems can be overviewed by a single person and the programs are reasonably small.

4. Functional Needs for Real-time Enhancements to Dartmouth BASIC

4.1. Introduction

The BASIC features which have made the language successful in different application areas also apply for a class of real-time applications. A great number of implementations already exist that use various extensions to meet real-time needs. This document is an attempt to define the common requirements of the real-time user and maintains the spirit of BASIC by using key words instead of parameterized calls.

The following diagram is an illustration of a possible real-time system:



Syntax examples used in this document are only for illustration and are not part of the proposal.

4.2. Process Input-Output

Process input-output functional interfaces allow access to specified analog and digital inputs and outputs. All the process input-output operations are executed in direct response to the occurrence in the program. (This classification is chosen because many comparison candidates use it.)

4.2.1. Analog input-output

The parameters needed for these operations are specifications of:

a) The hardware channel(s)
b) Access Mode (random, sequential, etc.)
c) Hardware-software conversion
d) Error information

The data transferred over the analog input-output interface will be represented in standard floating point form.

4.2.2. Digital (discrete) input-output

The parameters needed for these operations are specifications of:

a) Digital I/O point or series of points
   (discrete I/O channels)
b) Hardware-software conversion information
c) Error identification information.

The data transferred may be to or from a standard floating point variable or a bit string variable (see Section 6).

4.2.3. Pulse (train) input-output

The parameters required for these operations are specifications of:

a) Channel(s)
b) Hardware-software conversion information
c) Error identification information.

4.3. Access to System Time and Date

The current time must be available to real-time BASIC programs for use in calculations and logging. It must give date (year, month, day) and time information (hour, minute, seconds).

4.4. Extension of conventional I/O (terminals)

Process Control configurations are usually not limited to a single system console. System configuration often includes teletypes, line printers and CRT's. An example is the need to print test results on terminal printers local to the tests being performed. There is therefore a need to extend the PRINT and INPUT statements to include the specification of the target devices. Operator and device errors in response to INPUT statements shouldn't be fatal.

4.5. Extension of statements controlling program flow

Dartmouth BASIC includes control statements that allow the interruption of the normal sequence of execution of statements by causing execution of a specified program or program part rather than the next higher line number. There are two approaches for incorporation of this feature. For simple systems it can be handled on the interrupt level by a statement that connects an interrupt to a line number or subroutine (case a).

The other approach is to use BASIC as a software layer on top of a real-time process-control system. Recent implementations more often use the second approach (case b).

a) Real-time BASIC requires extension of these control statements to allow the execution of specified subroutines upon the occurence of real time events related to the system time "AFTER 10 seconds" "EVERY 100 minutes" or to the occurence of interrupts or special input commands.

All these events have to be connected with their related subroutines before their occurrence at run time. In such a simple system the introduction of different interrupt subroutine levels should be avoided.

Example:

```
  10   ON   INT  5  GOSUB   1000
1000  REM  "START OF SERVICE ROUTINE"
  .
  .
  .
       RETURN

  30  AFTER   T1   GOSUB  2000
2000   REM  "START OF SERVICE ROUTINE"
  .
  .
       RETURN
```

b) If BASIC is hooked onto a real time executive it clearly has to use the tasking mechanism of the underlying system. The programmer should only be faced with events and tasks. This also overcomes one of the BASIC restrictions namely having all variables global. Each task has its own line numbers and variable names only the task names and event names are global.

When writing a task the user types the task name and specifies the task level. The level of a task defines which tasks are allowed to interrupt it.

The synchronisation of tasks can be done by two operations on events namely.

WAIT event [,event...] and SIGNAL event

WAIT suspends the task until one of the events occurs. An "ACTIVATE task name" can activate other tasks. Deactivation must be done by the task itself.

example:

```
TASK  BASE  LEVEL  0
10  REM  BEGINNING  OF  TASK  "BASE"  LEVEL 0
     -
     -
     -
100  ON  HALLO  ACTIVATE  GET
           ↖
         eventname

200  END
TASK  GET  LEVEL  1
10  REM  HERE  STARTS  TASK  "GET"  LEVEL 1
     -
     -
     -
100  END
```

example of synchronisation

```
TASK  BASE  LEVEL  0
 10  REM  BEGIN
  -
  -
 50  WAIT  TICK
 60  REM  "TICK"  IS  A  DECLARED  EVENT
  -
  -
100  END

TASK  PUT  LEVEL  1
 10  REM  BEGIN
  -
 20  SIGNAL  TICK
```

For the transfer of data between tasks two commands are foreseen

```
          SEND  dataname VIA eventname TO taskname
and       RECEIVE  dataname  FROM  taskname.
```

example:

```
          TASK 1                              TASK 2
     100 SEND A(3) VIA YOU TO TASK 2          20 WAIT YOU
                                              30 RECEIVE B(2) FROM TASK 1
```

By this mechanism the element A(3) of task 1 is sent to B(2) of task 2 using event "YOU" as the synchronising element. (Additions for access to remote files see 9.)

Time events are often used and therefore should be treated in a different way. A minimum set is: "200 AFTER time interval ACTIVATE taskname" but also "AT time" would be desirable. EVERY causes some problems and therefore should be avoided.

For the treatment of events a "SET eventname" would be necessary. "TEST event name" or "CLEAR eventname" is not essential (done by the system).

The system clearly must be able to save and get tasks to and from bulk storage, list and delete tasks.

4.6. Extensions to Data Types and Operations

The number of data types in real-time BASIC should be kept to a minimum. The introduction of new data types may offer advantages (for example run time efficiency). In the proposed "Minimal BASIC"*, two data types exist. The data type floating point variable and the data type string. The corresponding data structure built of these data types are array (1 or 2 dimensional) and string arrays.

---

* Appendix III

Manipulation of analog and digital I/O data can be done using the standard floating point representation. Appropriate conversion routines in analog and digital I/O statements will convert discrete (digital) inputs-outputs into the internal machine representation. In this form, no integer representation will be required. However, a bit pattern data structure is strongly desirable, especially in applications like automatic digital testing where a large number of bits in the order of hundreds, are required. It must also be possible to write, input and print a representation of bit pattern e.g. in octal, hexadecimal or binary. In summary, the following process data have to have internal representation in the BASIC system:

1. Analog input-output - Values can be represented by standard floating point variable.

2. Digital input/output - The bit pattern of physical "0" or "1" combination can be represented by a string of "0" or "1" in a bit string or by an equivalent floating point representation. (i.e. right justified)

Additional operations are required to manipulate internal representation of external discrete process data. The Boolean operators needed are:

AND, OR, NOT, Exclusive OR, SHIFT, BIT TEST/SET/CLEAR

Preferrably functions should be avoided.

4.7. External Subroutines

Calls to user-generated non BASIC subroutines that are external to the BASIC system should be provided.

4.8. Extension to Error Handling

A very important consideration in the use of real-time BASIC in industrial applications is the type of error handling capability that should be provided in the language.

Whenever errors occur during BASIC program execution (such as overflow, incorrect data input, I/O errors, etc), they should result in the setting of an error flag using a system variable. Such an error flag may be tested by an extension of the control statement of the form:

```
5    ON ERROR   GOTO (GOSUB)   100
```

A scheme should be provided to allow the user to identify the error cause.

4.9. File Handling and Interprocessor Communication

There are strong requirements for file handling in measurement and control applications. It is important that the standard syntax for file manipulation could be extended to access files that may be resident on the mass storage of a remote computer in a distributed system network. The same remarks apply to segmenting, where program chains or tasks may be resident in the same computer mass storage device or in another computer in a distributed system network.

Summary of BASIC Statements

Appendix I

	Purpose		*Example*
1. Elementary BASIC			
INPUT	Reads data from teletype	17	INPUT A$, X
READ	Reads data from data block	17	READ X, Y1, M(J + 2, 3), N$
DATA	Storage area for data	17	DATA – 1, 2.07, 31416E – 4, 127829, JONES
PRINT	Types numbers and labels	17	PRINT "ANSWER ="; X, A * B, N$
LET	Computes and assigns value	17	LET X2 = X + Y ↑ 2
GO TO	Transfers control	17	GO TO 175
IF	Conditional transfer	17	IF T(I, J) < = 25 THEN 175
FOR	Sets up and operates a loop	17	FOR N = 10 TO 1 STEP – 1
NEXT	Closes loop	17	NEXT N
END	Final statement in program	17	END
2. Advanced BASIC			
INPUT	Reads data from the teletype	17	INPUT X, Y4, Z
LINPUT	Inputs an entire line as a single string	17	LINPUT A$
DEF	Defines a function	17	DEF FNG (X) = 2 * SIN (X) * EXP (– X)
FNEND	End of a multiple-line DEF	17	FNEND
GOSUB	Transfers to a subroutine	17	GOSUB 800
RETURN	Returns to statement following GOSUB	17	RETURN
RESTORE	Restores data to beginning	17	RESTORE
REM	Permits comments	17	REM BEGINNING OF SUBROUTINE
DIM	Declares dimensions of lists and tables	17	DIM A(12), B(3, 5)
STOP	Stops program	17	STOP
CHANGE	Convert string of characters to a vector, vice versa	17	CHANGE A$ TO V
ON	Multiple way branch	17	ON X+Z GO TO 200, 400, 750
RANDOMIZE	"Randomize" the random number generator	17	RANDOMIZE
3. File instructions			
FILES	Specified files	17	FILES DATA; *
FILE	Names or renames a file	17	FILE #2: N$
INPUT	Reads from a teletype file	17	INPUT #1: X, A$, Y
PRINT	"Prints" to a teletype file	17	PRINT #1: A, B, C
READ	Reads from a random file	17	READ #2: X, Y(1), Y(2)
WRITE	Writes to a random file	17	WRITE #2: R, S+T
RESET	Set random file pointer	17	RESET #2: LOC(2) – 1
SCRATCH	Scratch a file	17	SCRATCH #3
4. Matrix instructions			
MAT INPUT	Reads data, any number, from teletype	17	MAT INPUT V
MAT READ	Reads a matrix from the data block	17	MAT READ Z(M, N)
MAT PRINT	Types a vector or matrix	17	MAT PRINT A
MAT +	Matrix addition	17	MAT C = A + B
MAT –	Matrix subtraction	17	MAT C = A – B
MAT *	Matrix multiplication	17	MAT C = A * B
MAT ( ) *	Scalar multiplication	17	MAT C = (COS (X)) * A
MAT INV	Matrix inverse	17	MAT C = INV (A)
MAT TRN	Matrix transpose	17	MAT C = TRN (A)
MAT ZER	Matrix of all zeroes	17	MAT C = ZER
MAT CON	Matrix of all ones	17	MAT C = CON (15)
MAT IDN	Identity matrix	17	MAT C = IDN

5. Notes

Variables	X, Y7, A, A(X), B(A(X), 5), A$, B7$, N$(I), T$(A,B)
Operations	+, –, *, /, ↑
Relations	<, < =, =, >, > =, < >
Functions	**SQR, SIN, COS, TAN, ATN, LOG, EXP, ABS, SGN, INT, RND, ASC, LOC, LOF, NUM, TAB**

PRECEDING PAGE BLANK-NOT FILMED

Appendix II

Nine candidates were chosen. It is clear that this is not a complete set and that other candidates should be also taken into account.

List of candidates:

ADAPTS (Varian Extended BASIC)	/1/
BASEX (Phys. Inst. Univ. Freiburg)	/2/
BASIC/RT 11 (DEC)	/3/
F P L (FOXBORO)	/4/
INDUSTRIAL BASIC (DEC)	/5/
PROCESS BASIC (Unicomp)	/6/
RTE-B (HP)	/7/
KENT K90 BASIC	/8/
CASIC B Schlumberger	/9/

Some of the examples may be erroneous due to incomplete descriptions.

Criteria for comparison

To justify the name real-time-BASIC a language should be a superset of Dartmouth-BASIC (or in the future of "Minimal BASIC" (ANSI)).
The extensions are classified in the following way:

1. Extension of the algorithmic part
1.1. Mnemotechniques
1.2. Objects and operations
1.2.1 Text handling
1.2.2 Logic Operations
1.2.3 Bit Manipulation
1.3. Specification of peripherals
1.4. Format specification
1.5. File handling
1.6. Insertion of assembler routines
1.7. Segmentation of user programs

2. Process I/O
Objects (Philosophy of treatment)
2.1. Analog inputs
2.1.1 Input of a single value
2.1.2 Input of more than one value without channel switching
2.1.3 Input of a single value with channel switching
2.1.4 Input of multiple values with channel switching

2.2. Analog Outputs
2.2.1 Output of a single value
2.2.2 Output of more than one value without channel switching
2.2.3 Output of a single value with channel switching
2.2.4 Output of multiple values with channel switching

2.3. Digital Inputs
2.3.1 Single Bits
2.3.2 Words

2.4. Digital Outputs

2.4.1 Single Bits

2.4.3 Words

2.5. Pulse inputs

2.6. Pulse output

2.7. Connection of other peripherals

3. External Interrupts

3.1. Connection of a program segment with an interrupt

3.2. Disabling and enabling of interrupts

3.3. Interrupt levels (Nested Interrupts)

4. Treatment of time

4.1. Start of a program segment at an absolut instant of time

4.2. Start of a program segment after a specified time

4.3. Repetition of a program segment after a specified time

5. Parallel execution of program segments

6. Synchronisation of program segments

7. Error treatment

1.1. Mnemotechniques

ADAPTS:

like Dartmouth-BASIC

BASEX:

Variable names use 1 to 4 alphanumeric characters starting with a letter.

BASIC RT/11:

like Dartmouth-BASIC

FPL:

Variable names start with a letter. An arbitrary number of alphanumeric characters can be used.

INDUSTRIAL BASIC:

like Dartmouth-BASIC

PROCESS BASIC:

Variables and labels use the first 6 alphanumeric characters of an alphanumeric sequence starting with a letter.

RTE-B:

like Dartmouth-BASIC

1.2. Objects and Operations on these objects

used datatypes:

ADAPTS:	real
BASEX:	real, string
BASIC RT/11:	real, string
FPL:	real

INDUSTRIAL BASIC: real, string
PROCESS BASIC: real, 3 times integer (variable length)
RTE-B: real

1.2.1 Text handling

ADAPTS: no

BASEX:

Text handling uses string variables and ASCII- and Hexa decimal constants. Storage for string variables (var. name $) is reserved by the CHAR-statement, i.e. 1 CHAR AX$ (27).
The stored characters can be addressed by AX$ (N) (single character adressing), by AX$ all of them or by AX$ (N, M) a substring.
Operations for the string variables are:

NOT$	Inversion per bit
AND$	AND per bit
OR$	OR per bit
&	Concatenation

In addition to that all comparison operations can be applied.

BASIC RT/11:

String variables and ASCII constants similar to Dartmouth BASIC are used. Comparison and concatenation are available. In addition there are system functions for string variables and constants.

FPL: no

INDUSTRIAL BASIC:

like BASIC RT/11

PROCESS BASIC: no

RTE-B: no

1.2.2 Logic Operations

ADAPTS:

AND, OR, NOT are used. Real values express the values of the logical variables
(0 = false, ≠ 0 = true)

BASEX:

like ADAPTS

BASIC RT/11:

no

FPL:

no

INDUSTRIAL BASIC:

no

PROCESS BASIC:

no

RTE-8:

like ADAPTS

1.2.3 Bit manipulation

ADAPTS: no

BASEX:

is handled by string variables (see 1.2.1). The system function CHG is used for conversion between numerical- and string variables.

BASIC RT/11:

is handled by string variables. System functions for conversion and special tasks are available.

FPL:

no

INDUSTRIAL BASIC:

like BASIC RT/11

PROCESS BASIC:

Assembler level and some Macro instructions allow bit manipulation on all data types.

RTE-B:

Bit manipulation is done by system subroutines on "reals".
the following subroutines are used:

IOR	oring	bitwise
INOT	inversion	
IEOR	exclusive or	
IAND	anding	
ISHIFT	shift	
IBSET	set a bit	
IBCLR	clear a bit	
IBTST	test a bit	

1.3. Specification of peripherals

ADAPTS:

no

BASEX:

By DEV (number of unit) I/O units can be activated with INPUT and PRINT. By DEV(0) the system terminal is activated.

BASIC RT/11:

INPUT # n: and PRINT # n is used to activate I/O peripherals and bulk storage. n is the logical unit number.

FPL:

PRINT (n)and INPUT (n)(n = unit number) are used to activate I/O units.

INDUSTRIAL BASIC:

like BASIC RT/11

PROCESS BASIC:

The Macros DEG, DAG can be used to define I/O units. Those can be activated by ISC, OSC, Ein, Aus macros.
INPUT n and PRINT n (n = unit number) can also be used.
ISC, OSC: I/O of one character
EIN, AUS: I/O of an arbitrary amount of information

RTE-B:

PRINT #n and INPUT #n (n = unit number) activate additional I/O units.

1.4. Format specifications

ADAPTS:

The system function TAB(n) is used to set the typewriter into n-th position (only with PRINT statements)

BASEX:

Also TAB(n) and format specification FMT. The specifications are:

A	automatic format
F (n.m)	floating point without exponent
E (n.m)	floating point with exponent
n and m	define total number of characters and number of digits after point.

BASIC RT/11:

TAB(n) only with PRINT

FPL:

no

INDUSTRIAL BASIC:

like BASIC RT/11

PROCESS BASIC:

The AUS macro can be modified by

GLE floating point with exponent
GLF floating point
ITG integer
ZTB line feed and TAB
ZTI line feed and TAB for address list element
HEX Hexadecimal information
VTX variable text
TEX fixed text
NEZ n-times same character
FVI pointer to format list (indirect)
FEN pointer to format list or END

RTE-B: no

1.5. File handling

ADAPTS:

Background storage uses magnetic Tape. Statements are

OPEN Open file
CLOSE close file
PUT write into file
GET read from file
FLIST list all files

DELETE delete files
SAVE write core buffer into file
CLEAR clear all files
ASSIGN load core buffer from file

BASEX:

There are subroutines (activated by CALL)
CREA open file
KILL delete file
ALTR change name
LENG change length
PROT define read/write protection
OPEN open file and assign logical number
CLSE close file
GFB, GFBS read block into numerical or string field
PFB, PFBS write block into numerical or string filed

BASIC RT/11:

Two types of files are defined:
a) ASCII-sequential file
b) virtual-memory file

a) is used to store ASCII characters for buffered I/O and program sequencing (see 1.7)
b) is a data background storage for data fields and single elements can be accessed, read or written.

Statements:
OPEN open and specify a file
CLOSE close a file
PRINT,INPUT (see 1.3)
blulk storage is tape and/or disc.

FPL:

no

INDUSTRIAL BASIC:

Sequential files to store numerical and string data.

Statements

PRINT	write
INPUT	read
RESTORE	positioning
CLOSE	close file
IF END ... THEN	End of file check

PROCESS BASIC:

Background storage can be used like other I/O units and can be accessed via AUS, EIN macros (formatted, unformatted, buffered, unbuffered, sequential, random with or without conversion).

RTE-B:

Magtape can be accessed by the following subroutines:

MTTRD	read
MTTRT	write
MTTPT	positioning
MTTFS	controlling

The subroutines are activated with CALL.

1.6 Insertion of assembly routines:

ADAPTS:

Assembly routines can be inserted at system generation time. These subroutines are activated by CALL's.

BASEX:

At system generation time a) and b) can be loaded.

a) Assembly routines. These are called by CALL with 0-4 parameters (numerical or string)

b) System process variables (see 2.)
These are accessed directly or via the PUT-statement (see 2.)
It is possible to pass one numerical parameter.

BASIC RT/11:

like ADAPTS. Number of parameters only restricted by line length.

FPL:

Assembler routines must be installed when the system is delivered and can't be loaded later. (CALL)

INDUSTRIAL BASIC:

Assembler routines can be loaded at system generation time. They have to be declared by UDEF ... and then can be called like any other. System function (sometimes with dummy variables) (no CALL). 4 numerical and 2 string parameters can be passed.

PROCESS BASIC:

The assembler level and the macro assembler level and two of the three levels in the program. The levels can mixed.

RTE-B:

Installation and call like ADAPTS.

1.7 Segmentation of user programs

ADAPTS:

no

BASEX:

It is distinguished between ROOT-programs and segments. A single level tree structure is used. A segment can only be called in the root. The programs roots and segments are translated separately and are then combined into a package. Only the root program is coreresident with one segment at a time. All segments use the same overlay area. Variables in segments which are not used in the root are local. All variables in the root are global. Data can be passed via global variables and files.

Statements:

LINK name	call of segment (only by root)
END SEGMENT	End of segment (only in segment)

BASIC RT/11:

The OVERLAY statement provides a means for inserting parts of programs into a running program (from ASCII-sequential files) (see 1.5.)
Long programs can be divided by the CHAIN-statement. Whenever a CHAIN is reached the core buffer is cleared and the next program part is loaded. The data transfer between the CHAIN-parts is only possible via files.

FPL:

no

INDUSTRIAL BASIC:

CHAIN exists.

PROCESS BASIC:

Global (COMMON) and local (DIMENSION) naming is possible.
Different programs (in core or on background storage) can be linked.

RTE-B:

no

2. Process - I/O

Elements which are used for Process I/O are called process objects. Generally nearly all objects are assembler programs which specify addresses and procedures by names or parameters. There are implementations with one name for several process conncections and others where several subroutines specify different functions of one process connection.

INDUSTRIAL BASIC as an exception and does not use assembler programs for this purpose.
Some different program systems use different process objects. These are explained below.

ADAPTS:

Process objects are assembler subroutines with invariant functions. Their process addresses are defined by parameters. For example: input of process data uses two subroutines:
a) Transfer of data from process input to internal data field.
b) Transfer of data from process input to background storage.
There is no distrinction between analog and digital input. The user has to specify the process address by "channel" and "frame". In the following example ≠ 1 is an analog and ≠ 2 a digital input. When calling the subroutine DATAI (see 2.1.1) the statements look like:

≠ 1 100 CALL DATAI (0,A(1),10,1,0)
≠ 2 100 CALL DATAI (0,A(1),10,2,0)
(frame is underlined, channel = 0)

A similar subroutine is used for process outputs. Two special subroutines are used to treat pulse- and status word output.

The specified I/O subroutines are able to increment the internal addresses up to a specified value.

BASEX:

Process objects are assembler subroutines and a new type of variable - the process variable. This variable can be defined by the user (using EQUI, EQUO see 2.7.) or is defined for standard peripherals.

Process variables are syntactically treated like normal variables (except only one index). The index can be used to pass channel specification. As far as semantics is concerned the process variables are connected with the environment whereas the other varibles are related to the computer system.

It is distinguished between input and output variables. Input variables can be used everywhere in numerical expressions except on the left side of an assignment whereas output variables are only on the left side of an assignment. To treat process output variables which generate invariant signals one can use the PUT statement.

The following example shows 3 process variables

a) input from process

b) value calculated by program

c) output of constant value.

```
1 EQUI INPU(X)=%...%      101 PRINT NORM-INPU(K)
2 EQUO OUTP(X)=%...%      102 LET OUTP(I+5)=A(K)+EXP(Y)
3 EQUO PULS=%...%         103 PUT PULS

    definition              run time call
```

... represents machine code

BASIC RT/11:

Process objects are assembler programs. There are several subroutines with different procedures for one process peripheral. Some of the procedures are specified by parameters. 5 "modules" (module = subroutine package for one process peripheral) exist.

1. Buffer and field treatment
2. Analog to digital conversion
3. Real time clock
4. Digital output
5. Display

The example shows an analog data input where the "ADC" subroutine package transfers one value and "RTS" 50 values into a buffer. One parameter in "RTS" (underlined) defines the input strobe sequence.

```
100 CALL "ADC" (1,A)
100 CALL "RTS" (A,0,4,50,0)
```

(see 2.1.)

FPL:

The process object is a process variable to be defined in a DEF-command (see 2.1.1) and can be activated or deactivated by the program. All the specifications of addresses and procedures are defined by that DEF-command. The name of the process variable can be used in numerical expressions (with certain restrictions). If the "Task" corresponding to the process variable is activated it transfers data from or to the process.
The example shows the definition of an analog input and the transfer of a value into the mainprogram.

```
DEF: AIN: ANALOG:
... specifications like
channelnumber
transferperiod
conversiontype
limits
PID-control
etc.

mainprogram:   ACT: ANALOG;
               LET: A=ANALOG/4;
               DEACT: ANALOG;
```

INDUSTRIAL BASIC:

Process objects are assembler functions which either be defined by the user (then they must be inserted into the system and declared by UDEF (see 2.7.)) or system functions for standard peripherals. Syntactically these functions are treated like the others, i.e. EXP, SQR etc. but they are not allowed to be used on the left side of an assignment. This causes use of dummy variables on the left side. The data transfer and other specifications are done by parameters.

```
Analog input    LET A=ANI(0,10)
Digital output LET D=SDO(1,1,0)
```

(D = dummy, see 2.1., 2.4.)

PROCESS BASIC:

Process objects are macros defined by DEG, DAG.
(which means: Define Input Device, Define Output Device)

RTE-B:

A philosophy similar to BASIC RT/11 is used. Process objects are assembler routines which name a process connection and a procedure.

The parameters are used to specify the channel and transfer of measurement and control data.
The example shows analog input with sequential and random channel switching.
a) 100 CALL AISQV (5,1,A(K),E)
b) 100 CALL AIRDV (5,C(1),A(K),E)

2.1. Analog-Input

2.1.1. Input of a value

ADAPTS:

use of general input routine
DATAI (channel, field element, number, frame, period)
A floating point number is stored in the field.
10 CALL DATAI (0,A(1),1,0,0)

Another possibility is to use the subroutine DATIF with the same parameters as DATAI, but an integer value is stored into file #n (second parameter)

BASEX:

The process variable INA (channel) allows access to ADC
10 LET A=INA(0)

BASIC RT/11:

The system subroutine "ADC" (channel, variable) is used

```
10 CALL "ADC" (0,A)
```

Another possibility is to use the system subroutine "RTS" which controls the analog input timing.

FPL:

The peripheral has to be defined.

```
DEF: AIN: INA; definition of name (INA)
IRG: -10,10;   measurement range
ERG: 0,1000:GPM; definition of used unit
```

In the program this input can be activated or deactivated

```
ACT:   INA
LET: A=INA
DEACT: INA;
```

A variety of specifications concerning data conversion, limits etc. is possible.

INDUSTRIAL BASIC:

The system function ANI (channel, amplification) is responsible for the transfer.

```
10 LET A=ANI (0,100)
```

PROCESS BASIC:

Definition of the peripheral by the macro DEG, then data are acquired by INP.

Definition:

DEG n, program address

(n = input device number)

Assignment of a value:

INPUT u A

RTE-B:

Use of system subroutine for sequential access
AISQV (number of channels, first channel, field, error parameter)

```
10 CALL AISQV (1,0,A,E)
```

Another possibility is to use the system subroutine for random access. AIRDV (number of channels, reference field, field, error parameter). The reference field holds the channel numbers which have to be used sequentially.

2.1.2 Input of several data values without channel switching

Parameter specifications like 2.1.1

ADAPTS:

```
1Ø CALL DATAI (0,A(I),1,0,0)
```

BASEX:

```
1Ø FOR I=I  TO  1ØØ
2Ø LET A/I)=INA(Ø)
3Ø NEXT I
```

BASIC RT/11:

```
1Ø FOR I=1 TO 1ØØ
2Ø CALL "ADC" (Ø,A(I))
3Ø NEXT I
```

FPL:

Definition of process connection as in 2.1.1

```
ACT: INA;
LET: I=1;
1Ø  LET: A(I)=INA;
LET: I=I+1
IF  (I  1ØØ) GOTO: 1Ø;
```

(there is no FOR loop in this implementation)

INDUSTRIAL BASIC:

```
10 FOR I=1 TO 100
20 LET A(I)=ANI(0,100)
30 NEXT I
```

PROCESS BASIC:

```
DEG  n ...., program address
FOR I=1 TO 100
INPUT n A(I)
NEXT I
```

RTE-B:

```
10 FOR I=1 to 100
20 CALL AISQV(1,0,A(I),E)
30 NEXT I
```

2.1.3 Input of a value with channel switching

In this example 5 values should be read from consecutive channels starting with 1 and stored into A(1) to A(5)

ADAPTS:

```
5 FOR I=1 TO 5
6 CALL DATAI (I,A(I),1,0,0)
7 NEXT I
```

BASEX:

```
5 FOR I=1 TO 5
6 LET A(I)=INA(I)
7 NEXT I
```

BASIC RT/11:

```
5 FOR I=1 TO 5
6 CALL "ADC" (I,A(I))
7 NEXT I
```

FPL:

Like 2.1.1, but five definitions DEF:AIN: INA; are necessary and five assignments in the program are needed.

INDUSTRIAL BASIC:

```
5 FOR I=1 TO 5
6 LET A(I)=ANI(I,1Ø)
7 NEXT I
```

PROCESS BASIC:

```
DEG 1, program address
.
.
DEG 5,
FOR I = 1 TO 5
INPUT I A(I)
NEXT I
```

RTE-B:

```
5 CALL AISQ  V(8,1,A(1),E)
```

2.1.4 Input of multiple values with channel switching

In this example 100 values should be read from each channel of 1-5 and stored into a data field.

ADAPTS:

```
1 FOR I = 1 TO 5
2 CALL  DATAI (I,A(I,1), 1Ø0,Ø,Ø)
3 NEXT I
```

BASEX:

```
1 FOR I=1 TO 5
2 FOR J=1 TO 1ØØ
3 LET A (I,J)=INA(I)
4 NEXT J
5 NEXT I
```

BASIC RT/11:

```
1 FOR I=1 TO 5
2 FOR J=1 TO 1ØØ
3 CALL "ADC" (J,A(I,J))
4 NEXT J
5 NEXT I
```

FPL:

Definitions as in 2.1.1. The program activates the inputs and transfers the values (FPL allows only one-dimensional fields)

```
ACT: INA1;
ACT: INA2;
...
ACT: INA5;
LET: I=1;
10 LET: A1(I)=INA1;
LET: A2(I)=INA2;
...
LET: A5(I)=INA5;
LET: I=I+1
IF (I   1Ø1) GOTO 1Ø;
```

INDUSTRIAL BASIC:

```
1 FOR I=1 TO 5
2 FOR J=1 TO 1ØØ
3 LET A(I,J)=ANI(I)
4 NEXT J
5 NEXT I
```

PROCESS BASIC:

```
DEG 1, program address
.          .
.          .
DEG 5,
FOR I = 1 TO 5
FOR J = 1 TO 100
INPUT I A(I,J)
NEXT J
NEXT I
```

RTE-B:

```
1 FOR I=1 TO 5
2 FOR J=1 TO 1ØØ
3 CALL AISQV (1,I,A(I,J),E)
4 NEXT J
5 NEXT I
```

2.2. Analog Output

2.2.1 Output of a value

ADAPTS:

Output procedure DATAO syntactically like input procedure DATAI

BASEX:

Use of process variable OUTA (channel)

```
1 LET OUTA (Ø)=A
```

BASIC RT/11: no

FPL: no

INDUSTRIAL BASIC:

The system function ANO (channel, value) can be used. On the left side there is a dummy D.

```
1 LET D = ANO (Ø, 1Ø23)
```

PROCESS BASIC:

see digital output (2.4.2)

RTE-B:

The system subroutine AOV (channel, channel sequence, data field, error parameter) in connection with SGAIN (channel, amplification factor) can be used.

```
1 CALL AOV (1,K(1),A(1),E)
```

Another possibility is DAC (channel, value)

```
1 CALL DAC (1,10)
```

2.2.2 Output of multiple values without channel switching

see 2.2.1 and analog input

2.2.3 Output of single value with channel switching

see 2.2.1 and analog input

2.2.4 Output of multiple values with channel switching

2.3. Digital Input

2.3.1 Input of single bits:

ADAPTS:

no

BASEX:

There exists a process variable INB (bit number)

```
1 LET A=INB(Ø)
```

BASIC RT/11:

no

FPL:

Digital input module is specified for bitwise input
DEF: DIN: INPU; naming
CHAN: Ø
BITIN: INB(1); naming of the single bits.
after being activated the program can access the single bits.
ACT: INPU;
LET: A=INB(1)

INDUSTRIAL BASIC:

The system functions RDI (basis-bit, number of bits) and RDO (basis-bit, number of bits) allow the input of single external bits or the input of the last digital output values (RDO read the table which was used for the digital output).
1 LET A=RDI(Ø,1)

PROCESS BASIC:

Assembler

RTE-B:

data are acquired by
RDBIT (channel, bit number, variable)
1 CALL RDBIT (Ø,Ø,A)

2.3.2 Input of words

ADAPTS:

Digital data are acquired by DATAI and DATIF using channel and frame specifications.
(see 2.1.1)

BASEX:

By using the process variable INW (register number) 16 bit binary data and by using IND (register number) BCD oriented data can be transferred. The data are read as the mantissa of a real number with exponent Ø.
1 LET A=INW(3)

BASIC RT/11:

The system subroutine "DIR" (n, variable, control and status) reads a 16 bit word from an input register and stores the BCD coded value for n=Ø and the binary value for n=Ø in the variable.

1 CALL"DIR" (1,A,N)

Another possibility offers the subroutine "DRS" which reads a sequence of digital data directly or under clock control into a buffer.

"DRS" (buffer, n, number, clock mode, control-status)

FPL:

Definition of a process connection with activation like 2.3.1

DEF: DIN: INPU;

CHAN: Ø

measurement program:

ACT: INPU;

LET: A=INPU;

INDUSTRIAL BASIC:

The number of bits in RDI, RDO (see 2.3.1) is changed. The bit sequence which is read must not be longer than 12 bits (input register length)

1 LET A=RDI (25,12)

PROCESS BASIC:

1. Using the EIN-macro
2. Using the DEG-macro in connection with the INPUT statement. (see format specifications 1.4.) (see also analog input 2.1.)

RTE-B:

Use of system sub program RDWRD (channel, variable)

1 CALL RDWRD (1,A)

2.4. Digital Output

2.4.1 Output of single bits

ADAPTS:

no

BASEX:

Like in 2.2.1 using the process variable OUTB (bit number)

```
1 LET OUTB(1)=1
```

BASIC RT/11:

no

FPL:

specification of the digital output module for outputting single bits.

```
DEF: DOUT: OUTP;          naming
CHAN: 1;
BITOUT: OUTBIT(1);        naming of single bits.
```

measurement program

```
ACT: OUTP
LET: OUTBIT (1)=1;
```

INDUSTRIAL BASIC:

Uses digital output function SDO (bit number, number, bit pattern)
(left side with dummy)

```
1 LET D=SDO(1,1,Ø)
```

PROCESS BASIC:

Assembler

RTE-B:

Uses system subroutine WRBIT (channel, bit number, variable)

```
1 CALL WRBIT (1,1,A)
```

2.4.2 Output of words

ADAPTS:

Output with subroutine DATAO (see 2.2.1)

BASEX:

Uses process variable OUTW (register number) for binary and OUTD (register number) for BCD coded digital output.

```
1 LET OUTW (2)=A
```

BASIC RT/11:

The system program "DOR" (n, variable, control variable) sets for n=Ø or clears for n≠0 the specified bits. The output control variable then holds the value

```
1 CALL "DOR" (Ø,A,N)
```

FPL:

Uses digital output module

```
DEF: DOUT: OUTP
CHAN: 1;
```

measurement program

```
ACT: OUTP
LET: OUTP=A;
```

(only if A has an integer value)

INDUSTRIAL BASIC:

Uses output function SDO (see 2.4.1), dummy at left side.

```
1 LET D=SDO (12,4,5)
```

(bit pattern of 5 (decimal) is transferred into bit 12 to 15)

PROCESS BASIC:

1. Using the AUS-macro for formatted output (see format bit 1.4.)
2. Using the DAG-macro in connection with the PRINT statement

RTE-B:

The system subroutine WRWRD (channel, variable) is used

```
1 CALL WRWRD (1,A)
```

2.5. Pulse input

ADAPTS:

no

BASEX:

The standard peripherals contain a counter which can be read by process variable INC (number). The counter is started by ACTC (number) and stopped by HLTC (number)

```
1 CALL ACTC(1)
2 CALL HLTC (1)
3 LET A=INC(1)
```

BASIC RT/11:

no

FPL:

no

INDUSTRIAL BASIC:

no

PROCESS BASIC:

no

RTE-B:

no

2.6. Pulse output

ADAPTS:

Output of a pulse by using PULSE (pulse number)

```
1 CALL PULSE (1)
```

BASEX:

Using the timer pulses of defined length can be transmitted. The process variable OUTT (number) sets a timer which transmits a pulse of the desired duration after ACTT(n).

```
1 LET OUTT (1)=5Ø
2 CALL ACTT (1)
```

BASIC RT/11:

no

FPL:

no

INDUSTRIAL BASIC:

no

PROCESS BASIC:

no

RTE-B:

no

2.7. Connection of other peripherals

ADAPTS:

Using assembler routines (see 1.6.)

BASEX:

Firstly assembler routines can be coupled. Secondly by the statements EQUI, EQUO process variables (I = Input type, O = Output type) can be defined to fullfill the necessary machine code (Hexadecimal constants). (New process variables can be loaded into the system to save EQUI-EQUO)

line number EQUI variable name = machine code

BASIC RT/11:

see 1.6.

FPL:

see 1.6.

INDUSTRIAL BASIC:

see 1.6.

PROCESS BASIC:

With macros DEG and DAG also non standard peripherals can be defined.

RTE-B:

see 1.6.

3. External interrupts

Each implementation is checked for:

3.1. connection program parts
3.2. disable and enable
3.3. Interrupt levels

ADAPTS:

Interrupt treatment in BASIC and assembler is defined. No detailed description for further evaluation.

BASEX:

Connection of program parts

The statement ON INT interrupt number: allows two possibilities after the ":". Firstly a branch to a predefined program or secondly the direct execution of single statements. The interrupt number is kept until it is redefined. Each connected program must end with a STOP statement.

Disable and Enable

The statements ENAB interrupt number and
DISAB interrupt number
allow disabling and enabling at each point of the program.

```
1 REM MAIN PROGRAM
2 ON INT 1: GOTO 1ØØ
3 ENAB 1
...
1ØØ REM INTERRUPT SERVICE
11Ø ....
2ØØ STOP
```

Interrupt levels

There are several levels with hardware priorities. Interrupts of the same level can't interrupt themselves but are stored and executed using the algorithm "smallest number first". The implementation uses 2 or 4 levels.

BASIC RT/11:

no

FPL:

No program call on interrupt exists. For analog inputs alarm limits can be specified (high and low limits) which cause a message

```
DEF: AIN: INA;
CHAN: 1
PER: 5;
IRC: 1Ø, 5Ø
ERG: Ø,9ØØ,DEGF;
...
ALM;
HIAL: 75Ø;
LOAL: 5Ø;
...
```

The units are Ø to 9ØØ degrees Fahrenheit. The limits are 50 F (LOw ALarm) and 75Ø (HIgh ALarm).

INDUSTRIAL BASIC:

Connection of a program part

The statement CONTACT n THEN line number causes the jump to line number if the contact n changes. The service programs have to be finished with DISMISS.

Enable and Disable

There are no special statements. The interrupt is enabled by CONTACT n THEN line number. The service program can be halted and the connection can be solved by CONTACT Ø THEN line number. The line numbers must be the same in both statements.

```
1 REM MAIN PROGRAM
2 CONTACT 1 THEN 1ØØ
...
1ØØ REM SERVICE PROGRAM
...
16Ø DISMISS
```

PROCESS BASIC:

Connection of program parts

The interrupt definition, stack organisation and priority assignment must be done by the user. The macro DIT connects the program with the operating system. For intermediate storage and saving of the current machine states each priority level must have its own stack. (In the example the stack is defined at the end of the service routine.) Interrupt service programs must be finished with a VPE or RIT macro.

Enable and disable

The macro ISL bit pattern is used to enable and disable the specified levels.

```
    DIT Stacklist/ :Passing of stack to operating system
≠   MAINPROGRAM
    ISL ØØ         :Level Ø enabled (bit Ø set)
≠   INTERRUPT SERVICE
    INTERRUPT ...
              ...
    RIT
    STACK $ 2 INTERRUPT :Start address of service routine
          PSW +3Ø       :Stack level Ø (in this example 31 words long)

    STACKLIST BRI ØØ STACK
              Ø
              Ø
    ...
    (Here the start addresses of the levels must be stored. The sequence
    defines the priorities.)
    ...
    (STACK, STACKLIST and INTERRUPT are labels)
```

Interrupt levels

The user defines the priorities of the service programs with the sequence of the addresses in the "stacklist". 12 levels are possible.

RTE-B:

Connection of program parts

The interrupt source can be defined by SENSE (channel, bit, value, trap number). CALL MPNRM deletes all SENSE definitions. Coupling is done by TRAP trap number GOSUB line number. The service program is ended by RETURN. Priorities can be defined by SETP (line number, priority)

Enable and Disable

The statement CALL ENABL (line number) enables the program part from the specified line number (which got an interrupt by SENSE). CALL DSABL (line number) disables the program.

```
1 REM MAINPROGRAM
2 CALL MPNRM
3 CALL SENSE (Ø,1,1,1)
4 CALL SETP (1ØØ,5)
5 TRAP 1 GOSUB 1ØØ
6 CALL ENABL (1ØØ)
...
100 REM INTERRUPT SERVICE
...
200 RETURN
```

Interrupt levels

A scheduler program supervises all interrupts and time dependent calls (up to 16) and causes execution related to priority.

4. Treatment of time

Remark: In general the treatment of time correlated statements is either of type "AFTER interval DO" or "AT absolute time DO". Those statements have to cause access to a clock. Three examples show the time handling.

It is distinguished between:

4.1. Start at an absolute time

4.2. Start after an interval

4.3. Repetitive, time dependent execution.

ADAPTS:

Uses system subroutine
SCHED (hour) (min) (sec) (line number) which is of the "AT ... DO" type.
Accessing absolute time is done by TIME (hour) (min) (sec)

a) Start at an absolute time (14:3Ø:ØØ)

```
   1 CALL SCHED (14) (3Ø) (ØØ) (1ØØ)
```

b) Start after a time interval (45 sec)

```
   1 CALL TIME (H)(M)(S)
   2 REM CALCULATE NEW STARTING TIME = NOW + 45 sec
   3 LET S=S+45
   4 IF (S> 6Ø) THEN 1Ø
   5 LET S=S-6Ø
   6 LET M=M+1
   7 IF (M> 6Ø) THEN 1Ø
   8 LET M=Ø
   9 LET H=H+1
  1Ø CALL SCHED (H) (M) (S) (1ØØ)
  ...
 1ØØ REM SERVICE
  ...
 15Ø CALL RESUME
```

c) Repetitive time dependent execution
Start like 4.1. or 4.2. Up to 19:15:ØØ the service routine should be started every 15 sec.

```
13Ø REM SERVICE
11Ø CALL TIME (H) (M) (S)
12Ø IF (H  19 AND M  15) THEN 3ØØ
13Ø REM CALCULATE LIKE (B)
...
14Ø CALL SCHED (H) (M) (S) (1ØØ)
15Ø REM SERVICE START
...
3ØØ CALL RESUME
```

BASEX:

There is a statement
AFTER time interval : statement

The same statements like in ON INT ... are permitted. In this implementation the time activations are of lower priority than the interrupts. To access the system time one can use the system variables HOUR, MIN, SEC, MSEC.

a) Start at an absolute time (14:30:00)

```
1 REM CALCULATE INTERVAL FOR START
2 LET TIME = (60 * (14 *60 + 30))-(60 *(HOUR *60+MIN)+SEC)
3 AFTER TIME   1000 : GOTO 100
```

b) Start after an interval (after 45 sec)

```
1 AFTER 45000 : GOTO 100
...

100 REM SERVICE
...
200 STOP
```

c) Repetitive time dependent execution with time limit
Start like a) or b). Up to 19:15:00 every 15 sec.

```
100 REM SERVICE
110 IF HOUR = 19 AND MIN = 15 THEN 300
120 AFTER 15000 : GOTO 100
130 REM SERVICE START
...
300 STOP
```

BASIC RT/11:

no

FPL:

Statements for time control are WAIT, WAIT UNTIL and WAIT UNTIL .... OR.

Multi tasking allows the establishment of tasks into the WAIT state.
WAIT only stops the task where the WAIT is programmed.

Access of system time via HOUR, MIN, SEC system variables.

a) Start at absolute time (14:3Ø:ØØ)

```
DEF: TASK: SERVICE;
WAIT UNTIL: HOUR = 14;
WAIT UNTIL: MIN = 3Ø
...
DEACT: SERVICE

Main program
ACT: SERVICE;
```

b) Start after an interval (45 sec)

```
DEF: TASK: SERVICE;
WAIT: 45;
...
DEACT: SERVICE;
```

c) Repetitive time dependent execution
Start with ACT: SERVICE; up to 19:15:ØØ every 15 sec.

```
DEF: TASK: SERVICE;
waiting like a) or b)
2 IF (HOUR ≠ 19) GOTO: 3
  IF (MIN ≠ 15) GOTO: 3
                GOTO: 4
3 ...
  WAIT: 15
  GOTO: 2
4 DEACT: SERVICE;
```

INDUSTRIAL BASIC

The statement TIMER <inverval> THEN line number is of the type AFTER ... EVERY ... DO.

TIMER Ø THEN <line number> deletes definition. (like CONTACT Ø when handling interrupt). The statement COUNTER <number> THEN <line number> is like AFTER .. DO, but in this case a counter is set in hardware and then decremented until content = zero. This causes jump to specified line number. Therefore the time for a decrement must be known and offer the right order of magnitude to be used in the program.

Access of absolute time can be done by CLK (n).

(n = Ø read clock, n ≠ Ø set clock)

a) Start at absolute time (14:30:ØØ)

(Assumption one count ≙ 1 sec)

```
1 REM MAIN PROGRAM
2 LET Z=143ØØØ-CLK(Ø)
3 COUNTER Z THEN 1ØØ
...
1ØØ REM SERVICE PROGRAM
11Ø ...
...
16Ø DISMISS
```

b) Start after an interval (45 sec)

```
1 REM MAIN PROGRAM
2 COUNTER 45 THEN 1ØØ
...
1ØØ REM SERVICE PROGRAM
...
...
15Ø DISMISS
```

Repetitive time dependent execution with time limit. Start like a) or b). Every 15 sec up to 19:15:ØØ.

c)
```
1 REM MAIN PROGRAM
2 COUNTER Ø THEN 5ØØ
...
...
5ØØ REM SERVICE
51Ø TIMER 15 THEN 5ØØ
52Ø LET Z = CLK(Ø)
53Ø IF Z > 191ØØ THEN 57Ø
54Ø REM SERVICE PROGRAM
...
...
56Ø DISMISS
57Ø TIMER Ø THEN 5ØØ
58Ø DISMISS
```

PROCESS BASIC:

Only access to time intervals is possible. The macro IPZ (programmed interrupt after interval) causes delayed start of a priority level like AFTER..DO. Macro VPE causes leaving a program level for the defined time and then start again. An internal counter exists, which is incremented every three milliseconds. The examples do not show stack- and level assignment (as for interrupts).

Start at absolute time:

unknown

Start after time interval (45 sec)

```
≠ MAIN PROGRAM
  ISL ØØ2   1 level enabled
  IPZ + 1 45  Start level 1 after 45 sec
  ...
≠ SERVICE PROGRAM LEVEL 1
  LEVEL 1 ...
          RIT
  (LEVEL 1 = Label)
```

Repetitive time dependent execution with time limit

Start like b), execution every 15 sec.

```
≠  SERVICE PROGRAM
   BEGIN IF limit THEN END
       ...
       VPE + 15
       GOTO BEGIN
  END  RIT
```

R.E-B:

There are the system program TRNON (line number , time) and START (line number , time) like AT .. DO, AFTER .. DO. The system program TIME (variable) accesses absolute time. In addition priorities can be assigned. (see 3., CALL SETP)

a) Start at 14:3Ø:ØØ

```
1 REM MAIN PROGRAM
2 CALL TRNON (1ØØ,143ØØØ)
...
1ØØ REM SERVICE PROGRAM
...
15Ø RETURN
```

b) Start after 45 sec.

```
1 REM MAIN PROGRAM
2 CALL START (1ØØ,45)
```

c) Repetitive time dependent execution with time limit
Start like a) and b). Every 15 sec up to 19:15:ØØ execute.

```
1ØØ REM SERVICE
11Ø CALL TIME (T)  {T in sec after midnight}
12Ø IF T > 19*36ØØ+15*6Ø THEN 19Ø
13Ø CALL START (1ØØ,15)
14Ø REM SERVICE PROGRAM
...
...
19Ø RETURN
```

5. Parallel execution of programs

It already has been mentioned that multi programming is difficult with BASIC. But for some applications it may be useful to run a background program in BASIC. How the implementations deal with that is shown below. (The start should be caused by program not by clock or interrupt.)

ADAPTS: no

BASEX:

The statement START level number : line number starts parts of a program connected with fixed priorities. These priorities are lower than time- and interrupt levels.

```
1 REM MAINPROGRAM
2 START 1: 1ØØ
...
1ØØ REM BACKGROUND PROGRAM
...
19Ø STOP
```

BASIC RT/11: no

FPL:

Multitasking is possible by defining several tasks activating them in a common main program.

```
DEF: TASK: F1;
DEF: TASK: F2;
etc.
```

Activation in the main program:

```
ACT: F1;
ACT: F2;
...
```

INDUSTRIAL BASIC: no

PROCESS BASIC:

The macro IPT starts a defined level which must have been enabled by ISL.

(Priority- and stack assignments are left out)
(see 3)

```
≠ PROGRAM LEVEL Ø
  ISL ØØ2
  IPT +1
  ...
≠ BACKGROUND PROGRAM LEVEL ;
  ...
RIT
```

RTE-B: no

6. Synchronisation of program parts

This is necessary if continuation of a process depends on one or more events which are set by other processes.

ADAPTS:

Using WAIT time interval only allows pseudo synchronization

BASEX:

WAIT numerical expression causes GOTO to itself if expression = Ø (FALSE). If value ≠Ø (TRUE) the program executes the next statement.

Note that WAIT does not enable program execution for lower levels if executed in a higher level. Therefore this statement is only useful on the lowest level.

```
100 REM MAIN PROGRAM (wait for interrupt but only 1,5 sec)
105 ON INT 1: GOTO 1ØØ
11Ø ENAB 1
12Ø LET M=MSEC
13Ø WAIT FLAG = 1 OR MSEC = M + 15ØØ
14Ø DISAB 1
...
...
5ØØ REM INTERRUPT SERVICE FOR INT 1
...
55Ø LET FLAG = 1
56Ø STOP
```

BASIC RT/11:

The system subroutine "WAIT" (n) causes a halt up to occurrence of the event specified by n. This event is generated by hardware which consists of a clock and a one-bit input.

n=Ø: time interval

n=1: trigger signal

n=2: means n=Ø or n=1

```
12Ø CALL "WAIT" (1)
```

FPL:

The statements

```
WAIT <time interval>
WAIT UNTIL <condition> OR  <time interval>
WAIT UNTIL <condition>
```

cause a time- or condition dependent halt of the task.

Other tasks continue.

```
DEF: TASK: TEST;
...
WAIT  FLAG = 1 OR 15
...
(15 means 15 sec)
```

INDUSTRIAL BASIC:

no

PROCESS BASIC:

no

RTE-B:

WAIT <time interval> halts the program for the specified time. During this time no other program is running. Interrupts are only stored and are treated after the time interval (max. 30 sec). The waiting time is not precise.

7. Error treatment

When controlling processes it is important to avoid system stop in the case of an error but to treat the error by program.

ADAPTS:

no

BASEX:

It is possible to start an interrupt program in case of an error. The system variable ERR holds the cause of the error. An advanced implementation will use an ON ERR statement.

BASIC RT/11:

no

FPL:

no

INDUSTRIAL BASIC:

no

PROCESS BASIC:

no

RTE-B:

There is a parameter defined to allow error treatment by program. (in the example called E)

```
1ØØ CALL AIRDV (5, (1, V1, E)
1Ø1 IF E = ... THEN  ...
```

A "failure option" FAIL: GOTO line number can be connected with system subroutines. The system variable IERR tells the cause. FAIL cannot be used with BASIC statements.

```
1ØØ CALL TRNON (2ØØØ, 122536) FAIL: GOTO 1ØØØ
...
1ØØØ IF IERR(X)=1 THEN 11ØØ
1ØØ1 IF IERR(X)=2 THEN 12ØØ
(X = dummy)
```

Comparisons

1.1 Mnemotechniques

Variables start with a letter (but not FN) and contain up to 32 characters of letters, digits or decimal point.

1.2 Objects and Operations

Real, string and integer (Logical = integer). Integer and real are freely mixed.

1.2.1 Text Handling

Text handling uses string variables and ASCII literals.

String variables are identified by $ (ABC$).

String constants are identified by single or double quotes ('ABC', "ABC").

Comparison and concatenation are available.

System functions:- CHAR, LEN, GAP, VAL, DATE, TIME, LINE, SEG, STR.

1.2.2 Logic Operations

Logic operations are applied bit by bit to integers. FALSE = ∅ TRUE ≠ ∅.

The operations available are NOT, AND, OR, XOR, IMP, EQV.

Real variables are converted to integers.

1.3 Specification of Peripherals

Up to 255 logical channels may be used and are referenced by:-

```
PRINT #N, - - -
INPUT #N, - - -
```

where N = 1 to 255

1.4 Formal Specification

As in NCC standard. A string (expression, literal or line) defines the required format in terms of literal characters and moulds.

For example

```
A$ = "A=+###.##  B=+*.**  C= <###  D=+#.####↑↑↑↑"
PRINT USING A$, 20.2, 15.9, "XYZPQ", 15
```

produces

A= 20.2 B= 15.9 C=XYZP D= 1.5 E 01

The functions TAB, GAP and LIN are also used

1.5 File Handling

Two types of file exist, ASCII serial and random access.

i) ASCII serial (discs, readers, CRT's, etc)

OPEN name for INPUT / OUTPUT / EXTENSION AS FILE channel no.

CLOSE channel number

INPUT, PRINT and INPUT LINE operate on serial files

ii) Random Access (disc only)

CREATE name WITH n RECORDS OF m AS FILE k

OPEN name AS FILE k RECORD SIZE l

CLOSE channel no.

These files are accessed by:-

GET #n RECORD m INTO array

PUT #n RECORD m FROM array

All files have a device name and file name
i.e.

PR = paper tape reader

DK : FRED = file FRED on disc unit DK

1.6 Insertion of Assembly Routines

Any function or sub-program can be a machine code routine.

The routine is converted through the assembler and then a special program into a special type of BASIC line. In this form, the machine code functions or sub-program are manipulated in the same way as normal BASIC lines.

Additional system routines can be also incorporated.

1.7 Segmentation of Programs

No CHAIN facility.

An OVERLAY statement allows specified sub-programs (parameterised subroutines) to be assigned to a number of overlays.

The calling mechanism is automatic and transparent.

Sub-programs do not have to be in overlays. They operate on their own set of variables plus their formal parameters.

e.g.

CALL ABCD (X%, 22, A$, Y(,))

2 Process - I/O

All process variables are scanned and converted by other parts of the system and are not the responsibility of BASIC.

BASIC may access any process variable from the system data-banks.

Three types of process variables are recognised, variable, logical and item. There are three associated operators (V@, L@ and I@) and each process variable has a "point number" (an integer).

For example

```
X = A + V@20 + V@B% (K+2)

IF  L@K  GOTO - - -
```

Each process point can have a system name and system functions are provided to convert between point names and point numbers.

Process outputs are performed by the statements

```
        ITEM
STORE   VARIABLE  xxx  IN point name
        LOGICAL
```

Note that there are two operating modes. In one (development mode) any STORE statements cause a message to be printed giving the parameters of the statement but no output is performed. In the "formal mode" the statement is obeyed normally.

2.1 Analog - Input

2.1.1 Input of a Value

Analog values are accessed as real variables as shown in above

i.e.

```
X = V@ point number
```

2.1.2 Input of Several Data Values without Channel Switching

No special provisions.

2.1.3 With Channel Switching

No special provisions.

2.1.4 Multiple Values and Channel Switching

No special provision

2.2 Analog Output

Depending upon the system, analog outputs are performed by STORE VARIABLE or STORE ITEM statements.

2.3 Digital Input

2.3.1 Input of Single Bits

Whether a digital input is one bit or a word is implicit in its point number.

Hence, it is referenced by

X = L@ point number

2.3.2 Input of Word

X = L@point number

2.4 Digital Output

2.4.1 / 2:4.2 Single Bits or Words

A digital output is defined as a bit or word by its point number.

STORE LOGICAL xxx IN poiht number

2.5 Pulse Input

Pulse inputs will have been converted to variables by the rest of the system

X = V@ point number

2.6 Pulse Output

Not a normal form of output on K90.

Would be handled by

STORE LOGICAL - - -

2.7 Connection of Non-Standard Peripherals

All peripherals supplied with a system are fully supported and the system is device independant. Addition of a new peripheral by a user requires regeneration of the system normally.

3 External Interrupts

These are not directly controlled by BASIC. Any program in the system can be started by an interrupt as defined at system generation.

Priority and dead-time are a function of the priority assigned to the program and normal system multiprogramming constraints.

4 System Time

System routines are available to start any program:-

i) after a time delay

ii) at a specific time

eg

CALL START (program no., data, time, time units)

CALL SCHED (program no., data, hours, min, sec)

A program may be repetitively scheduled by an operator request. This may commence at any point within 24 hours and have a repitition rate between 1/256 sec and 24 hours in units of 1/256 sec.

A program may also be delayed by:-

CALL WAIT (time, time units)

5 Parallel Exectuion of Programs

K90 is a multi-programming system. At any time up to 127 programs may be time-sharing on a priority basis.

K90 BASIC has two modes, development and formal. In development mode the BASIC program is not permitted to affect the rest of the system and operates at a background level. In formal mode a BASIC program may have any priority (256) and interact with the system in the same manner as any other form of program.

Every program has a name and a number and system routines convert between the two forms.

Any program may start any other and pass one piece of information.

i.e.

CALL START (program name, data, Ø, Ø)

In addition, the start may be time dependant as shown in section 8.4

6 Synchronisation

No formal synchronisation mechansim is available.

7 Error Treatment

The system will supply default values for non-fatal errors (ie over-flow).

The user may trap any error by:-

ON ERROR GOTO - - -

Two variables define thererror.

ERTYPE = a number defining the error

ERLINE = the line number causing the error

On occurrence of an error,, the user may:-

i) Ignore the error and restart at the line causing the error or any other line

i.e.
RESUME at same line
RESUME 100

ii) Allow the system to handle the error as if ON ERROR GOTO has not been used

ie
GOTO SYSTEM

In the case of process information secondary information is available on the validity of the values.

CASIC - Schlumberger

CASIC is a single user conversational programming language for CAMAC peripheral using the standard statements of BASIC and some extensions -

1 - Extension of the algorithmic part

1.1. Mnemotechniques

Symbols use the first 6 alphanumeric characters of an alphanumeric sequence starting with a letter

1.2. Objects and operations

used data types :

- integer numbers - octal
  - decimal

i.e - if the special character @ is used the value typed in is for CASIC in octal

i.e. @ 2000 means 1024

- OCT and DEC functions can be used in a PRINT statement to output numbers in octal or decimal format

- real number

- text handling : NO

- bit manipulation : AND, OR, XOR, NOT, SHIFT are implemented by functions

i.e - AND (A, OR (B,C))

- logic operations are available with bit manipulation in conjunction with the IF statement.

.../..

1.3. Specification of peripherals

- for CAMAC : by

DCLCAM name of unit, ... (declaration statement)

and STATION name of unit = Camac address (assignment statement)

I/O can be activated in an arithmetic expression or not, using the name of unit.

- for non CAMAC peripherals by

PRINT or INPUT (available only for teletype)

1.4. Format specification

- for CAMAC : NO

- for teletype :

- input format is implicit, depends either on the type of the variable declaration, or on the character preceding the input value

- output format is either implicit, or using OCT, DEC, BTD, DTB functions.

1.5. File handling : NO

1.6. Insertion of assembler routines

assembler routines can be loaded at system generation time. They have to be declared by DCLFNT and then they can be called like any right function. 16 numerical parameters can be passed.

1.7. Segmentation of user programs : NO

2 - Process I/O

Handles only CAMAC peripherals
input and output are always for a single value in digital format
analog input or output, pulse input or output are available with the corresponding CAMAC module. All modules are handled in the same way.

For that purpose three types of peripheral statements :

1 - Declaration statement

A symbolic name must be assigned to each CAMAC register used in the program and any reference to a CAMAC register must be done with this symbolic name. The form is :

DCLCAM symbol, symbol

i.e. DCLCAM ADC, PULSE, SCALE(3), PREC

2 - Definition statement

the statement STATION is used to indicate the geographical position of a CAMAC register, the form is : STATION symbol = (B, C, N, A)

B for branch number

C for crate number

N for address in the crate

A for sub-address

```
i.e.    STATION ADC = (Ø, 1, 1, Ø)
        STATION PULSE = (Ø, 1, 2, Ø)
        FOR I = 1 TO 3
        J = I + 2
        STATION SCALE(I) = (Ø, 1, J, Ø)
        NEXT
        STATION PREC = (Ø, 1, 6, Ø)
```

3 - Execution statement

3.1 - OPERATE statement for CAMAC modules and CAMAC crate actions

3.1.a. to test, initialize, enable, disable modules or crates, nemonic symbols are

TSTL, CLRL, ENBL, DISBL for modules

and INIT, CLEAR, DISHIB, INHIB, ENABLE, DISABLE for crates

3.1.b. other OPERATE functions are executed with the general statement : OPER associated to a function value as an argument of the CAMAC register name -

i.e. OPER ADC (26) enable the CAMAC module register named ADC, since CAMAC function 26 means "enable".

3.2 - Read/write statement like in BASEX mentionning the CAMAC register anywhere in the program in conjunction with a CAMAC function code. The corresponding process is executed.

i.e. A = ADC (Ø) means 'read'the last value of the process variable ADC, with the CAMAC function Ø and 'store' this value in the normal variable A, and 'initialize' a new conversion if ADC is an analog digital converter as in this example.

The philosophy is very similar to BASEX however there are 32 kinds of CAMAC functions that can appeared and, in order to limit the number of declaration statements, the type of process (given by the CAMAC function) is defined at run time.

3 - External interrupts

3.1. Connection of a program segment with an interrupt

the statement: ONLAM name of the process variable DO line number, causes a jump to line number if the corresponding interrupt occurs -
the service programs must be finished with CONTINUE.

3.2. Disabling and enabling of interrupts

the external interrupts are handled with the CAMAC functions 8 (test interrupt), 10 (clear interrupt), 24 (disable interrupt), 26 (enable interrupt) using OPERATE statement or the corresponding mnemonics.

3.3. Interrupt levels unknown

4 - System time NO

5 - Parallel execution of program segments NO

6 - Synchronisation of program segments NO

7 - Error treatment

e statement: ON ERROR Do line number, allows the jump to the mentionned line number when the CAMAC X error occurs.

.../...

Appendix III

Literature

I. Firmen:

1. DEC	LAB 8/e Software BASIC/RT	
	OS/8 INDUSTRIAL BASIC	1973/74
	BASIC RT/11 Language Ref. Manual	1973
	GENDAC Ref. Manual	1971
	INDAC 8 Programmers Ref. Manual	1969
	INDAC 8 Operators Guide	1970
	IDACS 8 Application Note	
2. Dietz	BASEX:Benutzeranleitung	1974
3. Fisher Controls	PC^2 Language Primer	1971
	PC^2 Operating manual	1973
	PC^2 Language Ref. Manual	1973
	PC^2 Advanced Programming Manual	1970
4. Foxboro	FPL Language Description	1972
	FPL Users Guide	1972
	FPL System Product Specification	1972
5. Hewlett-Packard	Real-Time BASIC	1974
6. Krantz Computer	BASIC	
7. Siemens	BASIC 320	1973
8. Unicomp	Process-BASIC	1975
9. Varian	ADAPTS	1973
10. Kent	K90 BASIC	1973
11. Schlumberger	CASIC 11B	1975

II. Other Literature

zu BASEX: "Höhere Prozeß-Sprachen für kleinere Rechner - das Beispiel BASEX", Lecture Notes in Computer Science 12, S. 436-446, 1974, Springer-Verlag

Beschreibung der Prozeßsprache BASEX
Interner Bericht C-25, Fak. Physik, Uni Freiburg, 1974

zu Process-BASIC: "Process-BASIC - Ein Programmiersystem für Prozeßlenkung mit KLeinrechnern"
Lecture Notes in Computer Science 12, S.425-435, 1974
Springer Verlag

CASIC-11B: BASIC Extension for CAMAC, Schlumberger 1975

"Interpreters and CAMAC"
Extensions to BASIC, Tate,G.R., Dawson,W.K.
Interne Berichte vom Nuclear Research Centre, Univ. of Alberta, Canada

"Draft for Minimal BASIC", X3J2 committee 1974

"Specification for Standard BASIC", G.M. BULL, W. FREEMAN, S.J.GARLAND, NCC Publication

"Introduction to BASIC Programming", KEMENY, KURTZ

User's guide for writing good BASIC programs

The current trend in software is towards structured programming. This implies hierarchical top-down program design as well as clean definitions of the interfaces between parts of the programs. Reduction of programming errors, easier documentation, program maintenance and interface definition for parallel programming are the main advantages.

BASIC as language does not support structured programming but one can improve the quality of BASIC programs by following some general rules.

1. Modularity

   A) Break the program into functional blocks.
   (mainroutine - subroutines)

2. Simplicity

   A) Make the logical flow of the program as simple as possible (in a "FOR-NEXT" LOOP do not arbitrarily transfer control out of the loop with "GOTO's"). The same is true for subroutines.

   B) Use one line for entry into a specific subroutine and larger functional parts of the program.

3. Readability

   A) Make the logical flow of the program obvious from the listing. Don't use remarks to excuse obscure logic. Make the logic clear and use remarks to clarify the role of the logic within the complete program.

   B) Separate subroutines from the program on the listing by using blank REMARK statements.

   C) Use remarks to completely identify the function of each subroutine.

   D) Then in the listing write all of the subroutines last so they appear last. Write the main program so that it flows logically from its first line to its last. Backwards jumps should only occur at the end of a loop.

   Example:

```
LIST

TRIG-3    08/04/69  09:29

100  REM  ANGLE-SIDE-SIDE
110  LET P = 3.14159265
120  DEF FND(X) = X*P/180
130  DEF FNS(X) = SIN(FND(X))
140  PRINT "ANGLE", "SIDE", "SIDE", "THIRD SIDE", "CASE 2"
150  READ B, X, Y
160  PRINT B, X, Y,
170  LET S = X*FNS(B)/Y
180  IF S > 1 THEN 290
190  IF S = 1 THEN 320
195
200  REM  TWO POINTS OF INTERSECTION
210  LET T = S/SQR(1-S↑2)
220  LET C = ATN(T)*180/P
230  LET A = 180 - B - C
240  GOSUB 370
250  LET C = 180 - C
260  LET A = 180 - B - C
270  GOSUB 370
280  GO TO 150
285
290  REM  NO POINT OF INTERSECTION
300  PRINT " NONE", " NONE"
310  GO TO 150
315
320  REM  ONE POINT OF INTERSECTION
330  LET A = 90 - B
340  GOSUB 370
350  PRINT " NONE"
360  GO TO 150
365
370  REM  ANGLE-SIDE-ANGLE ROUTINE
380  LET Z = X*FNS(A)/S
390  IF Z >= 0 THEN 420
400  PRINT " NONE",
410  RETURN
420  PRINT Z,
430  RETURN
440
450  DATA  60, 10, 8,   60, 10, 9,   60, 10, 11
460  DATA  120, 10, 8,   120, 10, 9,   120, 10, 11
470  DATA  90, 3, 5,   60, 5, 5,   30, 10, 5
999  END
```

PRECEDING PAGE BLANK-NOT FILMED

SECTION V

REPORTS ON SOME EXISTING AND PROPOSED PROCEDURAL LANGUAGES IMPORTANT IN THE DEVELOPMENT OF THE LTPL

The six languages described herein have all been considered in the deliberations of the LTPL Committee. Of particular importance has been RTL/2 and PEARL. The pertinent Workshop Minutes references are as follows:

1. "A Language for Writing Real-Time Systems - LTR," Minutes of the Third Meeting of the Purdue Workshop on Standardization of Industrial Computer Languages, pp. 119-125.

2. "Some Proposals for RTL/2," Fourth Workshop Minutes, pp. 291-312, by J. G. P. Barnes

3. "PEARL, the Concept of a Process and Experiment-oriented Programming Language," Ibid, pp. 162-175, by J. Brandes, S. Eichentopf, P. Elzer, L. Frevert, V. Haase, H. Mittenderf, G. Müller, and P. Rieder.

4. "Industrial Programming Language - LAI," Ibid, pp. 145-266.

5. "Elementary Description of the Features of the PROCOL Language," Minutes, Seventh Workshop, pp. 205-209, 217-233, 251-263, by G. Louit.

6. "Development of a Process Control Language - PCL," Minutes, Ninth Workshop, Appendix V, pp. 230-237, by H. Kuwahara, T.Hayashi, K. Fukuoka, and T. Bandou.

PRECEDING PAGE BLANK-NOT FILMED

SESA

A LANGUAGE FOR WRITING REAL TIME SYSTEMS

L T R

1.- INTRODUCTION -

LTR uses Algol as background with special possib
for solving real time problems.

It is associated at run-time with a standard sup
(or monitor) written separately which deals with schedulin
the tasks and their environment : events, resources, input
output, "dynamic data".

2.- DATA -

2.1.- Elementary classes -

- arithmetic variables -real (floating point)
  -integer (short, normal, lo

- logical variables (or bit strings) (short, normal,
  with bit access.

- boolean variables

- character string variables with character access.

- quality variables (status variables) : variables t
  one of a limited set of different values declared
  symbolic names and called attributes.

  (These variables can be considered as an ex
  of boolean variables).

...

- Reference variables : variables used to access the elements of a set (indirect adressing)

(see below).

2.2.- Data structures -

- Arrays : static structures of same type data

- Set : dynamic structures.

Set declaration gives a template using classical variables declaration.

At run-time, elements corresponding to this template can be created and erased. Reference variables with their operators are used for accessing a given element of a set. The supervisor deals with these dynamic data (memory allocation, linkage).

2.3.- Real time data (or program control data) -

- Event : it is a concept with pulse property : its activation has meaning only for the taskawaiting it.

- Resource : it is a concept with state property, associated with a service. A service can be a device or a piece of software.

A service and its resource, can have one or several users. The resource can be used for control synchronization of this service.

.../...

3.- PROGRAM STRUCTURE -

An LTR program is a list of articles.

3.1.- Data articles -

In this type of article appear only declarations and their initializations. Data declared in data articles are global and can be used in any other article. All declarations can appear in it and set declarations can appear only in these articles.

3.2.- Procedure articles -

It is the same notion as subroutine in Fortran. Data declared in such an article are local. (Accesses limited to the internal blocks).

Use of real time tools is limited in this article because they are not reentrant.

3.3.- Process article -

Writing is formally thesame as for procedure article but implementation is different to get reentrant code. Local data and parameters belong to a task i.e. at each process activation (run-time allocation).

Data declared in a process article are local. All real time tools can be used and, interesting property, events and resources can be declared.

.../...

Dynamic event and resource are thus obtained.

3.4.- Inter program data communications -

- by global data (declared in a data article)

- parameters in procedure and process call (call by address or reference, call by value).

4.- COMMUNICATIONS BETWEEN AN LTR PROGRAM AND THE STANDARD SUPERVISOR -

4.1.- A process call creates a task with a given priority. This priority is used by the supervisor for task scheduling.

- OPENED Call : the calling task goes on.

- CLOSED Call : the calling task stops and will be terminated only when the called task is executed.

The supervisor deals also with events and resources for task scheduling.

4.2.- Supervisor deals with dynamic data for creating, erasing elements of a set and gives the LTR program reference value for accessing these elements.

4.3.- Event activation and event expression awaiting.

4.4.- Resource possibility : reserve, free, blocked, deblocked.

.../...

4.5.- Symbolic naming of an interrupt level. Association of a procedure article to this name. Control of the status of an interrupt level.

4.6.- Input - Output -

- Symbolic naming of a given address device.
- Input - Output commands with the following parameters :
  - symbolic name of the device
  - command to execute (quality)
  - buffer name
  - error boolean
  - resource name for control
  - logical variable name (bit string to deal with a possible error).

The supervisor deals with storage allocation.

CONCLUSION -

The supervisor is ready

- The compiler and system will be running about June 1970
- No overlay in this implementation
- Target language : assembly language of the CII Iris 50 computer (≃ 360.40).

.../...

- Supervisor size (without tables) = 10 K bytes
- Compiler size = 128K bytes.

SOME PROPOSALS FOR RTL/2

J G P Barnes

2 11 70

PRECEDING PAGE BLANK-NOT FILMED

CONTENTS

1 Introduction

2 Style
- 2.1 Brackets
- 2.2 Reserved words
- 2.3 Strings
- 2.4 Bytes

3 Procedure
- 3.1 Nesting
- 3.2 Inner blocks
- 3.3 Specifications
- 3.4 Function designators

4 Arrays
- 4.1 Lower bounds
- 4.2 Literal arrays
- 4.3 Instant arrays
- 4.4 Equivalence

5 Types
- 5.1 Address variables
- 5.2 Fixed point
- 5.3 Boolean
- 5.4 General

6 For statements

7 Switches

8 Environmental equiries

9 Data set initialisation

A1 Use of address variables

A2 Nesting example

A3 Questionnaire

PRECEDING PAGE BLANK-NOT FILMED

1 INTRODUCTION

Several implementations of RTL/1 have now been completed and considerable experience in the use of the language has been gained.

This note suggests some alterations which are probably desirable in order to improve efficiency and acceptability of the language.

Some of the suggestions made here might be considered an anathema to the Algol idealist; however we need a practical programming language and not a beautiful ideal.

The suggestions are not complete; it is hoped to issue further suggestions covering such areas as segmentation. This note is being issued now in order that the reactions of potential users may be judged.

A questionnaire is included in an appendix and readers might find this a suitable medium in which to express their comments.

2 STYLE

It is a fact of life that Algol based languages suffer badly when represented using commonly available character sets. The general legibility of RTL text as it appears on line printer listings and its convenience for typing would be improved by the following alterations.

2.1 Replacement of square brackets by round

Square brackets are not essential for the analysis of RTL text although they are helpful to the human eye. However, the EBCDIC character set currently available on System 4 and 360 line printers does not have them and we have resorted to the use of ¦ and ¬ which is to say the least do not improve legibility.

Note also that some programming languages do not logically distinguish arrays from functions anyway; Precedent: POP-2 TPL PL/1 FORTRAN.

2.2 Reserved words

As interactive computing increases more and more highly qualified staff find themselves typing program text. It is thus most desirable that this should be as painless as possible.

Our experience with various formats has indicated that the current style (e.g. .BEGIN) is preferable to either quote inclusion (e.g. 'BEGIN') or underlining (e.g. KEY9 begin). However the . symbol is in shift case on teletypes (although alphabetic on 029 card punches), and thus not very convenient.

It is suggested therefore that the various language words be reserved and not available as identifiers. It might be agreed that this would render text less legible: perhaps a standard conversion routine could put % symbols in - they would be ignored by the compiler.
Precedents: PL/1, FORTRAN, POP-2, TPL.

2.3 Strings

The symbols available to represent pairs of string quotes are not very satisfactory. The current %[and %] are difficult to read. The existence of ' suggests that we might as well follow PL/1 and use the same symbol for opening and closing quotes.

An embedded quote would be represented by a pair of quotes as in PL/1. To avoid compiler difficulties, a warning could be issued if a newline occurred in a string.

Spaces in strings will be denoted by the underline/break character (ISO 5F). This is printed variously as

← ... _ ⌐

2.4 Bytes

The type char will be renamed byte. This is to emphasize the fact that a byte is merely an unsigned integer (≤ 255) and need not necessarily denote a character.

3.1 Procedure nesting

The current implementation allows nesting to a total depth of 4. FORTRAN allows only 1 and is known to be inadequate. An excessive allowance for nesting implies inefficiencies in implementation and is one of the problems with Algol 60 (which allows an infinite number).

Experience suggests that whereas 4 levels have proved useful they are not necessary. It is suggested that 2 levels be considered. Separate control routines for external and internal procedure could be provided, these would be considerably fasterthan the present routines, the code of all procedures would be a word shorter and the stack space required would be 2 words less per link cell.

3.2 Inner blocks

RTL/1 has not allowed blocks except as procedure bodies. This is partly historic because it was not allowed in SM 3. (An inner block is like a compound statement but with declarations). In

order to obtain the full benefit of address variables (see 5.1) and for general flexibility it is proposed that inner blocks be allowed.

It is to be noted that inner blocks not containing array declarations can be handled very efficiently at run time by treating the declarations as if they were at the outer level for all purposes other than scopes. Such an inner block would not then count as a level as far as the nesting restriction is concerned.

3.3 Specifications

Delete value which is the common case and use ref or name to denote the by-name parameter.

Precedents: Algol 68, IMP.

3.4 Function designators

Programmers often mis-use function designators within the corresponding procedure declaration. A construction like

RETURN (<expn>)

wouldavoid the necessity for the compiler to allocate space for the designator and allow more flexibility. The effect would be to evaluate the expression and exit from the procedure with the value as result.

In non type procedures the statement

RETURN

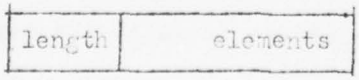
would be allowed to denote exit.

Precedent: PL/1, POP-2.

4 ARRAYS

4.1 Lower bounds

A serious inefficiency in RTL (compared with SNL say) is the need for a pair of words to handle arrays. This makes parameter sequences rather long. The following proposals enable a single word to suffice and still retain the security features of RTL which have been valuable: The lower bound of all arrays to be fixed. Each array will then be stored thus



and the single base address will then be used to access the elements and the length. A useful bonus will be the ability to access the length of an array in the language by an expression of the form

length < array identifier >

this will save the passing of the length as a separate parameter when necessary.

Multi-dimensional arrays will be stored truly as arrays of arrays. The handling of sub arrays can remain as now. Note that the length operator can be applied to any substructure of a multi-dimensional array.

It will be noted that in the case of one dimensional arrays the present security features are adequate - that is check on store only. However in the case of storing multi-dimensional array elements all the bounds must be individually checked and not the final address only. Similarly checking must be applied when extracting slices. This is obviously going to make some programs a bit slow but I think that multi-dim arrays are not used to such a large extent that this is too serious.

Finally there is the question of strings. The above proposal would lead us to abandon the read only protection currently provided. I do not think that this is serious. No code can be corrupted - only the string itself. Note also that since the length of the string will now be available the terminating 255 can be omitted; this has always seemed a curiosity anyway. The compiler will no longer be able to share strings unless identical; I doubt if the loss will be enormous.

In summary:

PROS

(i) Simpler parameter passing

(ii) Length operator available

(iii) More compact structure in 1 dim case

(iv) 255 anomoly in strings removed

(v) More compact coding in 1 dim case

(vi) Shorter compiler

CONS

(i) Lower bound fixed

(ii) Less compact structure in multi dim case

(iii) Slower storage in multi dim case

(iv) No protection for strings

(v) Less compact coding in multi dim case

Once it is established that the lower bound be fixed the question arises should it be 0 (like Autocode) or 1 (like Fortran)?

In favour of 0

(i) More natural for systems programming in code but not sure about high level languages.

(ii) Freedom to start at 1 and merely waste a location (but extravagant to do this for matrices)

(iii) Probably faster subscript checking

In favour of 1

(i) More natural for numerical analysis

(ii) No confusion between length and upper bound of array

4.2 Literal arrays

Currently strings are the only instance of literal arrays and the concept should be extended to all types. An integer literal array thus (1, 2, 3)

A real one thus (1.0, 2.0. 3.0)

A multi dimensional one thus

((1, 0, 0), (0, 1, 0), (0, 0, 1))

or ('pig', 'horse' 'dog')

and note that the latter is not rectangular.

These arrays would reside in the coding of the procedure (like strings now) and should only be read from.

Also if we have array variables (see 5.1) then one would be able to say

```
int array () x;
x:= (1, 2, 3)
```

but of course only the first level storage is on the stack.

4.3 Instant arrays

An instant array is somewhat analogous to a literal array and provides a means of setting up an array dynamically without having to name it.

Thus it might be convenient to hand over an instant array as parameter. We could denote an instant array as a set of expressions thus

(p, q, r + 4)

might be an instant 1-dim integer array.

Since an instant array is of length known at co pile time, it can be set up in first level stora e.

Some confusion is possible between literal and instant arrays, perhaps they shouldonly be allowed as para eters when the confusion is likely to be minimal.

4.4 Equivalence

It is proposed that an equivalence feature be introduced whereby a sequence of scalar vairables may be mapped onto an array of the same type.

e.g. int x, y, z, i, j, k alias a;
then a(4) would refer to i.

This enables one to perform block transfers of grou s of scalars and also to change (e.g. by a conversational program) one of them by a dynamic reference whilst retaining legibility and efficiency for normal specific access.

This could be extended to arrays thus

int array (3) p, q, r, s alias t;
then q(1) is t(2, 1)

5 TYPES

5.1 Address variables

Some recent languages (and in particular Algol 68) allow the explicit manipulation of addresses, and the declaration of address variables. The use of such variables requires care in a block structured language; it is only too easy to take addresses out of scope. But nevertheless address variables used correctly can bring considerable economies in coding.

Something along the following lines is proposed.

The concept of a type is extended to allow the declaration of v riables whose value is the address of other (normal) variables. The format of the declaration will be identical to the expression of bynane parameters (which are address variables)

BEST AVAILABLE COPY

AD-A036 455 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

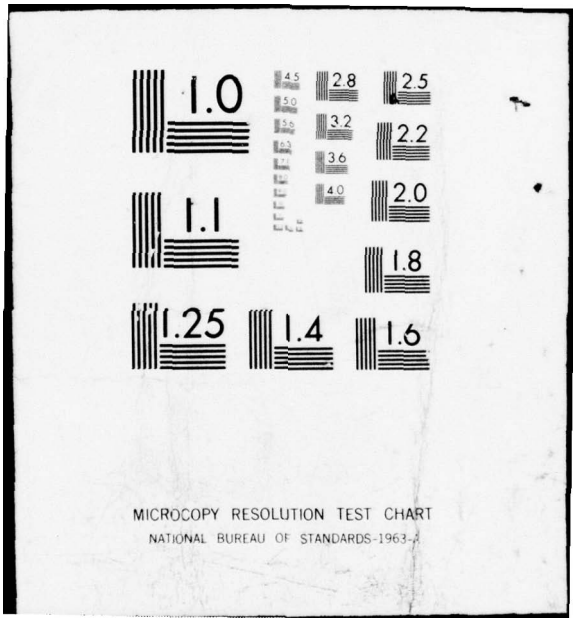
UNCLASSIFIED NL

3 OF 3
AD
A036455



END

DATE
FILMED
3-77


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

e.g. int name p, w, r;

As far as assignment statements are concerned these vari
behave like the formal parameters already in RTL. Thus i
variable occurs on the LHS then the allowed form on the
the same as currently allowed for an actual parameter an
use in expressions on the RHS is as for formal parameter
This, of course, means that the actual formal correspond
parameters is purely the mechanism of assignment. The l
therefore unified.

It will be appreciated that this need not complicate the
much, the intermediate codes all exist anyway, only phas
re organising.

Note as a corollory that it will now be possible to have
procedures,

e.g. int name proc p();

Similarly one wishes to declare arrays with only the firs
storage (as in current formal parameters) or with dopes
only p rt way.

e.g. int array (,) x;

could set up the first level storage only; x must be a 2

int array (5,) y;

would set up the first level storage and the 5 holes for
level of array storage but not the elements themselves.

Note that I am assuming fixed lower bounds and have move
list to before identifiers - this simplifies compilation

int array (5,5) z;

would set up all the storage in the normal way.

Also e.g. int name array (2,) p;

Assignments

Assignments to array variables and the values of array va
are unambiguous and pose no problems. However, in the c
scalar variables there are 6 possible types of assignmen

Suppose we have int name n, m;
int u, v;
then we wish to perform

BEST AVAIL

(i) c(v):=c(u)

(ii) c(c(n)):=c(u)

(iii) c(c(n)):=c(c(m))

(iv) c(u):=c(c(n))

(v) c(n):=u

(vi) c(m):=c(n)

These are currently represented in RTL as follows:

(i) by v:=u and parameter mechanism

(ii) by n;=u

(iii) by n:=m

(iv) by u:=n

(v) by parameter mechanism

(vi) by parameter mechanism

and we now wish to express (v) and (vi) in assignments.

I suggest the assigment action be the action across parameters; thus the type of value to be assigned is determined by the destination and in order to assign indirectly we use the operator val. The RS is then interpreted appropriately so (following Algol 68).

(i) v:=u

(ii) val n:=u

(iii) val n:=m;

(iv) u:=n;

(v) n:=u;

(vi) m:=n;

Alternatively one could be quite specific and have

(i) v:=u

(ii) valn:=u

(iii) val n:=val m

(iv) u:=val n

(v) n:=ref u

(vi) m:=n

The above is very explicit and is a bit like CORAL in some respects. (Less chance of erros but perhaps a bit tedious to use).

A third approach (T.P.L) is to add a new type of assignment (bit ad hoc I feel)

(i) - (iv) As now

(v) n<= u

(vi) m<= n

For example of the use of address variables see Appendix 1. (The explicit convention has been used).

Typical uses of address variables are in manipulating pointer to buffers, matrix algebra etc.

e.g. instead of a [i, j] :=a [i, k];

write b:= a [i]; b [j]: = b[k]
(where array b has been declared).

Security

There are two main problems (i) ensuring that uninitialised address variables are not used (we have ignored this problem in ALT/1 with section variables) and (ii) ensuring that addresses are not used outside their scope.

Two solutions spring to mind: (a) only allow address variables in a 'system' version of the language and leave the risks to the programmer, or (b) restrict their use so that invalid actions cannot result. Solution (a) needs no investigating so consider (b).

Initialisation

Either the variables are automatically initialised to safe dummies or the user must specify initialisation. To do the latter in a declaration will mean introducing inner blocks otherwise the full benefit of the address variables will not be felt. One does not wish to be forced to initialise address variables to a useless value because the useful value has not yet been computed. See also 3.2

Scoping

No address variable may be used if its value is out of scope. The simplest way of preventing this is to ensure that the value of an address variable is in scope whenever the variable itself is; this implies that assignments must be restricted to passing addresses between variables in the same block or from an outer block to an inner one.

One consequence of this is that an address variable in a data set can only refer to variables in that data set.

Precedents; Algol 68, C RAL 66, TPL

5.2 Fixed point arithmetic

The current version of RTL and SML3 have facilities for scaled fixed point arithmetic. To my knowledge these have never been used; one serious attempt was made but lack of knowledge of ranges or intermediate values caused us to resort to real arithmetic.

The facilities provided by the compiler in this area are largely a convenience, it is possible to use integer arithmetic and perform the scaling oneself - this is easy because RTL allows shift operations. It is therefore proposed to delete scaled integers. This would greatly simplify the compiler.

Shifts

The direction of a shift is often known but its size is not. At the moment there is no way of informing the compiler of the direction alone. The addition of operations SHLA, SHRA, SHLL, SHRL is proposed.

5.3 Booleans

Boolean variables and values in the true Algol sense are elegant but can lead to inefficiencies in commonly occurring circumstances unless a complex (and therefore unreliable)compiler can be used.

Thus in RTL/1 clauses like

if s = 1 or s = 2 then

are poorly compiled.

A convenient method of overcoming this is to delete the concept of a Boolean type from the language and introduce instead the "condition" used in an if chuse. Logical operations on integers considered as bit patterns would remain.

Thus it is proposed that a structure similar to that of CORAL be adopted.

The operations and, or $< \leq > \geq = \neq$ would only be used within conditions and a condition would be evaluated from left tto right only as far as is necessary to determine the result.

Logical operations equivalent to and or and not for integer variables would also be available but with a different notation to avoid confusion. The actual names chosen for CORAL (differ, union, mask), are not universally liked and alternative suggestions would be welcome.

Precedent: CORAL 66, POP-2

BEST AVAILABLE COPY

5.4 General Types

There are times when conventional t pe dependent high level languages do no have the required flexibility. Note our own difficulties with I/O.SML3 had to resort to special statements for this purpose.

In order to overcome these difficulties in a general and legal manner the general type is proposed. A variable f type general can contain any value plus a flag indicating the type of the value. Thus it is likely to be moderately inefficient. Operations will also be needed to identify the type of the value. Having identified the type of a value one will probably wish to transfer the value to a variable of the appropriate type. This can be conveniently performed by the "conforms to and becomes" of Algol 68.

Thus supposing we have a procedure with one parameter which could be an integer value or a one dimensional real array. The coding might be as follows:

```
proc p (gen x);
begin int i; real array () a;
   if i::=x then action if integer else
   if a::=x then action if array else
   fail ...
```

the clause

```
i::=x
```

has value true if the types are compatible and in this event the value is transferred.

Note that ::= unfortunately already has an established meaning as a meta symbol; Algol 68's choice is thus not liked and an alternative is sought. =&:= or =:= have been suggested.

It would probably be best to allow no operations on general v riables other than the above and assignment.

Using general variables and instant arrays it would be relatively easy to provide arbitrary record type I/O like SML3 or FORTRAN.

```
e.g. proc write (gen array ()a);
     begin ... end
```

used thus write (('answer is', x, 2, 0))

BEST AVAILABLE COPY

6 FOR STATEMENTS

It is proposed that a style of statement similar to Algol 68 and PL/1 be adopted. The full statement is

```
for i:= a by b to c while d do S;
```

The effect is to iterate as usual with the additional priviso that d must be true. The iteration is terminated by either the count reaching or passing the limit, by d becoming false or by jumping out.

The control variable is a variable local to the loop and is read only. This gives the compiler better opportunities for optimisation.

Various parts of the statement can be omitted.

`by 1` if the step is 1

`while d` if d is always true

`to c` if the loop is to be terminated by the while or jumping out.

e.g. `for i:= 1 to 10 do ...`

`for i:= 1 by 1 while p(i)<q(i) do ...`

The whole of the iterative part can be discarded thus

```
while check (x) do ...
```

Finally if the control variable is not used in the statement (as e.g. in the RTL/1 statement)

```
for i:= 1 step 1 until 100 do newline ();
```

this can be contracted to

```
to 100 do newline();
```

(or would you prefer `for 100 times do newline();`)

The omission of the control variable when not used will make possible the rapid compilation of better coding in these cases.

Precedents: PL/1, Algol 68

BEST AVAILABLE COPY

7 SWITCHES

Switches have proved to be somewhat anomolous in behaviour and in fact there is currently a restriction preventing their use non locally. The inability to declare switches within for statements coupled with the rule that one may not jump into for statements has meant that programming has not always been as transparent as desirable.

A solution is to have a switch statement rather than a switch declaration. e.g.

goto case i of L1, L2, L3 else .... ;

Precendent FORTRAN, Algol 66, Algol 68

N B: An alternative nomenclature could be considered

e.g. execute i of ....
branch
select
switch

8 ENVIRONMENT ENQUIRIES

Following P Naur it is proposed that some information about the object machines characteristics be made available to the programmer.

intbit	number of bits in integer representation
intbytes	number of bytes which can be stored in the same space as an integer
realbytes	number of bytes which can be stored in the same space as a real
wordbytes	number of bytes which can be stored in the same space as a reference

The above are effectively macros

Also the procedure (supervisor call?)

intproc stackfree()

will provide the number of bytes of stack not occupied.

Precedents: Algol 66, Algol 68

BEST AVAILABLE COPY

9 DATA SET INITIALISATION

This could be optimal or mandatory with default values.

For scalars it is proposed that the declarations be thus

```
int x:=3, y:=7;
```

For arrays thus

```
int array (3) p := (1, 2, 3), q:= (4, 5, 6);
```

and the upper bound may optionally be replaced by an asterisk when the value can be obtained from the initial list.

In particular it must be omitted if we allow non rectangular arrays thus

```
int array (*,*) t:=((1,2, 3), (1, 2), (1))
```

and a typical example might be an array of messages

```
byte array (7,*) days:= ('sunday','monday'...)
```

where the individual messages are of different lengths.

JGPB/JP
3 11 70

A1 Address variables

This example illustrates

(i) How a typical procedure looks in RTL/1

(ii) How it might look transliterated into RTL/2

(iii) The use of address variables in RTL/2 to avoid unnecessary subscript evaluations and thereby speed inner loops

R T L 1

```
%PROCEDURE INVERT(%REAL %ARRAY 2 A; %REAL %ARRAY B; %VALUE %INTEGER N);
%BEGIN %INTEGER I,J,K;
       %REAL S,C;
       %FOR J:=1 %STEP 1 %UNTIL N-1 %DO
       %BEGIN C:=0;
              %FOR I:=J %STEP 1 %UNTIL N %DO
              %BEGIN %IF ABS(A[I,J])>C %THEN
                     %BEGIN C:=ABS(A[I,J]); K:=I %END
              %END;
              S:=B[K]; B[K]:=B[J]; B[J]:=S;
              %FOR I:=J %STEP 1 %UNTIL N %DO
              %BEGIN S:=A[K,I]; A[K,I]:=A[J,I]; A[J,I]:=S %END;
              A[J,J]:=1/A[J,J];
              %FOR I:=J+1 %STEP 1 %UNTIL N %DO
              %BEGIN S:=A[I,J]*A[J,J];
                     B[I]:=B[I]-S*B[J];
                     %FOR K:=J+1 %STEP 1 %UNTIL N %DO
                         A[I,K]:=A[I,K]-S*A[J,K]
              %END
       %END;
       B[N]:=B[N]/A[N,N];
       %FOR I:=N-1 %STEP -1 %UNTIL 1 %DO
       %BEGIN %FOR J:=I+1 %STEP 1 %UNTIL N %DO B[I]:=B[I]-B[J]*A[I,J];
              B[I]:=B[I]*A[I,I]
       %END
%END
```

```
PROC INVERT(REAL ARRAY(,) A, REAL ARRAY() B);
BEGIN INT K,N;
      REAL S,C;
      N:=LENGTH B;
      FOR J TO N-1 DO
      BEGIN C:=0;
            FOR I:=J TO N DO
            BEGIN IF ABS(A(I,J))>C THEN
                  BEGIN C:=ABS(A(I,J)); K:=I END
            END;
            S:=B(K); B(K):=B(J); B(J):=S;
            FOR I:=J TO N DO
            BEGIN S:=A(K,I); A(K,I):=A(J,I); A(J,I):=S END;
            A(J,J):=1/A(J,J);
            FOR I:=J+1 TO N DO
            BEGIN S:=A(I,J)*A(J,J);
                  B(I):=B(I)-S*B(J);
                  FOR K:=J+1 TO N DO A(I,K):=A(I,K)-S*A(J,K)
            END
      END;
      B(N):=B(N)/A(N,N);
      FOR I:=N-1 BY -1 TO 1 DO
      BEGIN FOR J:=I+1 TO N DO B(I):=B(I)-B(J)*A(I,J);
            B(I):=B(I)*A(I,I)
      END
END
```

NOTE (i) Absence of % and shorter forms of reserved words

(ii) Abbreviated for statements

(iii) Third parameter omitted since length can be obtained from length operator

(iv) Square brackets replaced by round

(v) i and j declared by occurance as control variables of for statements.

RTL/2 with address vbls

```
PROC INVERT(REAL ARRAY(,) A, REAL ARRAY() B);
BEGIN INT K,N;
      REAL S,C;
      N:=LENGTH B;
      FOR J TO N-1 DO
      BEGIN REAL ARRAY() AJ:=A(J);
            REF REAL BJ:=REF B(J), AJJ:=REF A(J,J);
            C:=0;
            FOR I:=J TO N DO
            BEGIN IF ABS(A(I,J))>C THEN
                  BEGIN C:=ABS(A(I,J)); K:=I END
            END;
            S:=B(K); B(K):=VAL BJ; VAL BJ:=S;
            FOR I:=J TO N DO
            BEGIN S:=A(K,I); A(K,I):=AJ(I); AJ(I):=S END;
            VAL AJJ:=1/VAL AJJ;
            FOR I:=J+1 TO N DO
            BEGIN REAL ARRAY() AI:=A(I);
                  S:=AI(J)*VAL AJJ;
                  B(I):=B(I)-S*VAL BJ;
                  FOR K:=J+1 TO N DO AI(K):=AI(K)-S*AJ(K)
            END
      END;
      B(N):=B(N)/A(N,N);
      FOR I:=N-1 BY -1 TO 1 DO
      BEGIN REF REAL BI:=REF B(I);
            REAL ARRAY() AI:=A(I);
            FOR J:=I+1 TO N DO VAL BI:=VAL BI-B(J)*AI(J);
            VAL BI:=VAL BI*AI(I)
      END
END
```

A2 Nested procedures

An example of 3 levels

```
task a;
begin     proc b  (....);
          begin   proc p();
                  begin.
                        .
                        .
                        .
                        goto L
                  end;
                  .
                  .
                  .
M:                .
                  .
                  transfer (p,....);
                  .
L:                < fail> ; goto M
          end;
          .
          b(....);
          b(....);
          b(.....);
end;
```

(i) b is a procedure which does some form of I/O and incorporates error recovery by reissuing the request if it fails. b has been made into a procedure because it is wished to use it many times.

(ii) transfer is an external procedure which performs the I/O using a parameterless procedure to perform streaming. The actual procedure used is p.

(iii) p must be embedded within b since p detects the error and wishes to transfer control to the label L in b in this event

(iv) b must be internal in order that data local to a can be accessed.

This example indicates a need for 3 levels if the strategy outlined here is to be used. It could perhaps be argued that the strategy is incorrect.

BEST AVAILABLE COPY

A3 It would be of great value to us if you could spare the time to complete the enclosed questionnaire and return it to:

J G P Barnes
RTL Project
Imperial Chemical Industries Limited
Central Instrument Research Laboratory
Bozedown House
Whitchurch Hill
READING RG8 7PF
Berks
England

Further copies of the questionnaire are available on request.

BEST AVAILABLE COPY

QUESTIONNAIRE ON RTL/2

Please complete or delete (*) as appropriate. The numbers adjacent to the questions refer to the relevant paragraph in the main document.

NAME:

ADDRESS:

INTEREST IN RTL/2: *(As a user of RTL/1
(As a potential user of RTL/2
(General

2 RTL/1 text is not very legible *agree/disagree

2.1 Square brackets should be replaced by round *agree/disagree

2.2 I prefer *(reserved words
(%BEGIN etc

2.3 Bearing in mind the characters available -
I prefer *(different symbols) for opening and closing quotes
(same symbols)

2.4 Prefer *char/byte

3.1 How many levels of nesting are necessary ? [ ]

3.2 Inner blocks *would/would not be useful

3.3 I agree that value is confusing *Yes/No
I prefer *ref/name

3.4 I prefer *RETURN/ as it is now
I suggest ..... as alternative word for RETURN

4.1 Which of the following statements best expresses your attitude to fixed lower bounds?

*a/b/c

(a) Ridiculous not to allow them to vary despite inefficiencies that might arise;

BEST AVAILABLE COPY

Questionnaire contd.

(b) Somewhat regret not allowing variable lower bounds but the advantages seem worthwhile;

(c) Always felt that lower bounds should be fixed anyway.

The lower bound should be *0/1

4.2 Literal arrays *Would/would not be useful

4.3 Instant arrays *Would/would not be useful

I*am/am not confused by literal and instant arrays

4.4 Some sort of equivalence feature is desirable. Do you think that the proposal described goes far enough? *Yes/No

5.1 I *agree/disagree that address variables are necessary

Which of the types of assignment do you prefer?

*((i) somewhat like Algol 68
((ii) the explicit form
((iii) the ad hoc extension

The restrictions on the use of address variables to ensure security might be such as to make them useless in some areas.

*Agree/disagree

If you agree do you think that a system version of the language in which restrictions are relaxed is sensible? *Yes/No.

5.2 I think that scaled fixed point variables are

*(useless in practise
(possibly useful
(essential

Directed shift operations *would/would not be useful

5.3 Which of the following best expresses your attitude to Boolean types *a/b/c

(a) I think RTL/1 is O K

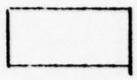
(b) Boolean types should be retained but the definition of the method of evaluation of boolean expressions should be changed to specify left to right evaluation only as far as is necessary

(c) The proposals to delete Booleans and institute conditions instead is preferred.

Questionnaire contd

5.4 **General** variables *do/do not seem necessary.

I propose the alternative symbol to ::= for the 'conforms to and becomes' operation .....



6 In general I *like/dislike the proposals for for-statements

I prefer *(to 100 do) to denote repetition
(for 100 times do)

7 I prefer *(switches as they are in RTL/1
(the case statement proposed

I suggest the following alternative nomenclature .....

8 I wish to have the following available to me *intbits/intbytes/realbytes/wordbytes.

I would also like .....

9 I prefer *optional/**mandatory** initialisation

If you prefer optional initialisation should there nevertheless be a default value? *YES/NO

10 This space is for general comments

Hauptschriftleiter: **Dr. P. Schmitz, Köln**
Redaktion: Dipl.-Volkswirt W. Eicken

Wissenschaftlicher Beirat:
Dr. H. Christen, Hamburg
Dr. E. Glowatzki, Darmstadt
Dr.-Ing. F. R. Güntsch, Bonn
Prof. Dr. W. Haack, Berlin
Prof. Dr. H. Herrmann, Braunschweig
Prof. Dr. W. Kämmerer, Jena
Dr. F. J. P. Leitz, Freiburg/Breisgau
Dr. K. Tjaden, Böblingen
Dr. rer. nat. R. Veelken, München
Prof. Dr. A. van Wijngaarden, Amsterdam

elektronische datenverarbeitung

Fachberichte über programmgesteuerte Maschinen und ihre Anwendung

Heft 10/1970
12. Jahrgang

This paper was first published in Elektronische Dataverarbeitung and may be copied only with their special permission.

PEARL*)

The Concept of a Process- and Experiment-oriented Programming Language

J. Brandes[1]), S. Eichentopf[2]), P. Elzer[3]), L. Frevert[4]),
V. Haase[1]), H. Mittendorf[5]), G. Müller[6]), P. Rieder[5])

Summary: The main features of a programming language for on-line-control of industrial processes and scientific experiments are described. Mainly discussed are the connections between the program and the process environment, the management of storage space and parallel activities, the synchronization of tasks, and the non-standard input-output. The paper describes structure and semantics of the language, but does not yet give an exact and definitive syntax.

Zusammenfassung: Dieses Konzept beschreibt die wesentlichen Eigenschaften einer Programmiersprache der Mittelebene für Anwendungen in der industriellen Prozeßsteuerung und in der Experimentiertechnik. Es werden hauptsächlich die Verkopplung des Programms mit der Prozeßumwelt, die Speicherverwaltung, die Steuerung und Synchronisierung der Parallelarbeit und nicht standardmäßige Ein-/Ausgabemöglichkeiten beschrieben. Der vorliegende Bericht beschreibt Struktur und Semantik der Sprache, nicht aber eine exakte Syntax, die noch von der "PEARL"-Arbeitsgruppe erarbeitet werden wird.

1. Introduction

In recent years it has been realized by many people that higher level programming languages like those for commercial or scientific applications will be advantageous and even necessary also in the fields of process and experiment automation [1, 2, 3]. Some proposals, such as RTL [4] or an extension of PL/1 [5] also proved the fundamental possibility of expressing specific characteristics of process and experiment programs by a higher level programming language.

The possibility of implementation had been doubted for quite a long time; however, it has been demonstrated meanwhile by the existence of some solution applicable to partial fields [6, 7].

1.1. Fundamentals of the "PEARL" Concept

"PEARL" is a compilable programming language of the intermediate level, such as ALGOL or PL/1, since this type of language offers maximum flexibility in addition to easy handling and learnability. Since a relatively large

*) Process and experiment automation realtime language

[1]) GfK-DVZ, Karlsruhe
[2]) AEG-Telefunken, Konstanz
[3]) Physikalisches Institut III der Universität Erlangen
[4]) Hahn-Meitner-Institut für Kernforschung, Berlin
[5]) Siemens AG, Karlsruhe
[6]) Fa. BBC, Mannheim

number of language elements are required in order to be able to treat also more complex problems, it is intended to define self-contained, fully upward compatible subsets for smaller computing systems. A brief summary is given of both the main features of a program for process applications, which practically prevent its being written in a conventional programming language (FORTRAN, ALGOL 60, COBOL, etc.), and means of solution proposed by the "PEARL" concept.

1.1.1. Machine Dependence

In a computer program for process application also the entire peripheral equipment must be described and selected which, in most cases, may be of non-standard design and more diversified than in commercial or scientific applications of electronic data processing. Due to the improbability of standardizing all possible hardware configurations and describing them uniformly at the level of an advanced programming language, a "PEARL" program is subdivided in principle into a system dependent part, the `system division`, and the `problem division` which is largely independent of the system. The `system division` provides the link between the process peripheral equipment and the program, whilst the `problem division` constitutes the control or measuring program proper. Thus, only the `system division` has to be modified when passing from one type of computer to another (Chapter 3.).

1.1.2. Parallel Activities

Usually a process program during operation splits up into a varying number of parallel "tasks" [8] some of which are performed completely independent of each other while others are correlated to each other, i.e., they have to be synchronized. The "task" is considered to be dynamical, i.e., it consists in the successive processing of one or several pieces of code. The process program is characterized by the interaction between the tasks which is made possible by synchronization with the help of semaphores [9] (Chapter 4.1.-4.4.).

1.1.3. Running Time and Storage Management

In a "normal" computer program, the aggregate computing time in principle does not matter. It is sufficient that the program will be terminated within a period of time reasonable for the user. However, since a process program, in most cases, must meet preset reaction times which, in the most extreme cases, may be of the order of a few cycle periods (e.g., application in nuclear physics experiments) the programmer must have more influence on the object code than he has in conventional programming languages.

A related problem is the management of storage allocation. Since often parts of the program code will be stored on some backing storage medium, the time for loading the code must be included into the running time of a task. The programmer must have the possibility to act on the time of loading as well as on the size of the code pieces to be loaded, though the working storage allocation shall be automated as far as possible (Chapter 4.5.).

1.1.4. Input/Output

In process programs there must be much more possibilities of input and output than in programs for commercial, scientific, or technical applications. A concept describing input and output was developed which will allow the handling of nearly all the input and output operations encountered in process technique under uniform aspects (Chapter 5.).

1.1.5. Types of Data and Language Features

In this respect, the familiar programming languages (ALGOL 60 and FORTRAN) prove to be unsufficient. Therefore it became necessary to include in the language more complex types of data, such as lists or structures. It has been attempted to combine all the elements of modern programming languages which are required for process purposes (PL/1, ALGOL 68), but to eliminate too expensive and rarely used types and, if necessary, replace them by new elements [10, 11] (Chapter 2.).

1.1.6. Relations to the Operating System

A programming language allowing interventions into functions of the operating system necessitates also a certain standardization of the properties of the operating system, e.g., uniform treatment of the interrupts. To obtain efficiency and satisfactory compatibility of process computing programs in a higher level programming language, the definition of certain minimum requirements will be needed which must be met by an operating system for a process computer.

1.2. Mode of Description

It has intentionally been avoided in this concept to propose an exact syntax of "PEARL". In exceptional cases where this is done, however, the usual notation of PL/1 is adopted [12].

2. Basic Language

2.1. Fundamentals

2.1.1. Character Set

The character set consists of approx. 60 characters including the latin alphabet, the decimal digits and certain special characters which are in common use.

2.1.2. Delimiters

Delimiters consist either of sequences of $n\,(n \geq 1)$ special characters (except blanks), or of sequences of $m\,(m \geq 2)$ letters (keywords). How or whether keywords are to be reserved, i.e. identical identifiers will have to be forbidden, depends on the exact definition of the language structure and, therefore, cannot be decided now.

2.1.3. Comments

A comment may be placed at every location in a program where a blank may be written; this comment will be enclosed by special characters as in PL/I.

2.1.4. Block Structure and Procedures

The language has a block structure similar to that of ALGOL 60 and PL/I with its consequences, especially regarding the scope of declarations.

Procedures will be treated similarly to those in ALGOL 60; to simplify matters, however, certain restrictions must be observed, e.g. regarding call by name of parameters. All formal parameters must be specified (as in ALGOL 60), [13].

2.1.5. Identifiers and Declarations

Identifiers are sequences of alphanumeric characters in which the first character is a letter. Identifiers, in general, refer to data (values) of a certain type (variables).

All identifiers used must be declared in the program; the scope of the declaration is the block in which the declaration is contained. Identifiers can also be declared, in certain cases, in the system division (see 3.), where the scope of the declaration is the entire program. Initialization of identifiers in declarations is possible.

Declarations not necessarily stand at the beginning of a block. If, however, an identifier is to be initialized in its declaration, this initialization must be dynamically executed before the identifier is used for the first time.

Besides identifier declarations, there are also type declarations (see 2.2.2.2.) and operator declarations (see 2.3.2.).

2.2. Types

There are certain basic types from which, according to specific rules, compound types can be formed.

2.2.1. Basic Types

Basic types are INTEGER and REAL of different precision, BINAL (bit string) of different length, STRING (character string) of different length, SEMA, FILE and ADDRESS. A type BOOLEAN or LOGICAL is not required, as it can be represented with BINAL of length 1.

The declaration symbols and the forms of constants will be decribed in the following sections.

2.2.1.1. INTEGER and REAL

Declaration symbols: INTEGER(n) and REAL(n) where n is the minimum number of decimal digits to be comprised within the mantissa. If (n) is missing, an implementation-dependent stadard length or precision will be assumend. By writing REAL(n), the next implemented mantissa length, which is not less than (n), will be taken.

Constants of the type INTEGER will be specified as a sequence of n ($n \geq 1$) consecutive decimal digits. Constants of the type REAL will be specified, similarly to PL/I, as decimal constants which are not integer constants. Blanks within one number are not permitted. The internally used mantissa length is the next implemented one which is not smaller than the number of digits in the constant.

2.2.1.2. BINAL

Declaration symbol: BINAL(n) where (n) is the number of bits. If (n) is missing, an implementation-dependent standard length will be taken.

In arrays and structures BINALs will be closely packed within limits of the effective addressability, in order to save storage capacity. For BINALs not in arrays and structures, the next implemented bit string length, not less than (n), will be taken.

Constants of the type BINAL can also be specified in octal notation with the digits 0 to 7, and in sedecimal notation with the digits 0 to 9, and the letters A to F.

Example: '110100101011'B binary
'6453'O octal
'D2B'S sedecimal

The notation of the BINAL constants is not as yet finalized.

2.2.1.3. STRING

Declaration symbol: STRING(n)
where (n) is the maximum number of characters contained in a string.
STRING constants will be enclosed in apostrophies.
Example: 'ANTON+BERTA'

2.2.1.4. SEMA

Declaration symbol: SEMA
A variable of the type SEMA can only assume integer values and is used to synchronize tasks (see 4.4.).

2.2.1.5. FILE

Declaration symbol: {SEQUENTIAL|RANDOM} FILE (*unit identifier*, pages/file, lines/page, characters/line)

FILE identifiers occur as parameters in I/O procedures. There are two types of FILEs, the SEQUENTIAL FILE in which only sequential reading and writing is possible and the randomly addressable RANDOM FILE.

FILE specifies data organized in pages, lines and characters and stored on external storage [11]. The declaration contains, therefore, the name of the unit (e.g. drum, disc), the number of pages per file, lines per page and characters per line.

2.2.1.6. ADDRESS

Declaration symbol: ADDRESS

Variables of the type ADDRESS have as values ADDRESS constants, i.e. identifiers (also subscripted) of variables which must not be of the type ADDRESS.

There are three address levels:

Level 2: ADDRESS variable, i.e. identifiers which indicate values of the type ADDRESS;
↓
Level 1: ADDRESS constants, i.e. identifiers which indicate values excluding those of type ADDRESS;
↓
Level 0: Values which are not of the type ADDRESS.

The following addressing mode is used for assignment: The left side of an assignment, i.e. the side to which the right side is assigned, must be on level 1 or 2 of the address levels mentioned above. The level of the right side of an assignment, i.e. the side which will be assigned, must come directly under the level of the left side.

Hence:

Left side	Right side
Level 2	Level 1
Level 1	Level 0

Accordingly, the level of the right side will be automatically adjusted to the specified level of the left side, as far as this is possible.

Example:

```
REAL A, B;
ADDRESS POINTER 1, POINTER 2;
```

```
A := 5.7
   /* LEFT: LEVEL 1; RIGHT: LEVEL 0; CLEAR. */;

B := A
   /* LEFT: LEVEL 1;
      RIGHT: NEXT ALSO LEVEL 1, AFTER ADJUST-
      MENT TO LEFT SIDE LEVEL 0, I. E. THE VALUE
      (HERE 5.7), INDICATED BY A, WILL BE
      ASSIGNED TO B. */;

POINTER 1 := A
   /* LEFT: LEVEL 2;
      RIGHT: LEVEL 1;
      THE ADDRESS-VARIABLE POINTER 1 ASSUMES
      IDENTIFIER A AS VALUE. */;

POINTER 2 := POINTER 1
   /* LEFT: LEVEL 2;
      RIGHT: NEXT ALSO LEVEL 2, AFTER ADJUST-
      MENT TO LEFT SIDE LEVEL 1, I. E. THE VALUE
      (HERE A), INDICATED BY THE POINTER 1 WILL
      BE ASSIGNED TO POINTER 2, AFTER WHICH
      BOTH ADDRESS-VARIABLES INDICATE A. */;

POINTER 1 := 3.9
   /* LEFT: LEVEL 2;
      RIGHT: LEVEL 0;
      FALSE, BECAUSE BOTH SIDES ARE NOT AUTO-
      MATICALLY ADJUSTABLE. */;

VALUE POINTER 1 := 3.9
   /* LEFT: LEVEL 1 BECAUSE OF THE OPERATOR
      VALUE APPLICABLE TO THE ADDRESS-VARIA-
      BLE; RIGHT: LEVEL 0;
      EFFECT: AS FOR A := 3.9, BECAUSE POINTER 1
      INDICATES A. */;
```

Also see example under 2.2.3.

Further details regarding the handling of ADDRESS variables are not, as yet, finalized.

2.2.2. Compound Types

2.2.2.1. Arrays

Arrays consist of elements of the same type. This type can be a basic type with the exception of SEMA and FILE, or a structure.

An example of a declaration for a two-dimensional array with elements of the type REAL and with the identifier MATRIX is as follows:

(5:100, 1:7) REAL(10) MATRIX

When transposing arrays to storage, the last subscript, i. e. the subscript at the extreme right-hand location, will be transposed one at a time ("line" transposition).

An element in an array will be addressed by an array identifier and a subscript list.

Example: MATRIX(7*J, 3) /* E. G. J=4 */

Furthermore, sections of an array can be addressed, which consist of more than one element.

Examples: 1) Call of the entire second column of the matrix declared above:

MATRIX(, 2) or MATRIX(*, 2)

2) Call of a section of the second column of the matrix declared above:

MATRIX(8:N, 2) /* E. G. N=55 */

According to this, the whole matrix could be addressed by

MATRIX(,) or MATRIX(*,*)

the array identifier alone is, however, sufficient for this purpose.

Examples of array constants:

1) Constant vector with three components:
(6, 7, 9)

2) A matrix

$$\begin{pmatrix} 3 & -5 \\ 7 & 9 \end{pmatrix}$$

will be written as:
((3, -5), (7, 9))

2.2.2.2. Structures

Structures will be formed from elements of (not necessarily) different types. The type of a particular element can be a basic type with the exception of SEMA and FILE or a structure or an array.

Structure identifiers will be declared similarly to those in ALGOL 68, whereby the structure type is used either directly as the declaration symbol, or by using a declaration symbol which has been explicitely declared in a type declaration.

Example: Declaration of an identifier (e. g. C) for complex values:

1st possibility:
```
STRUCTURE (REAL RE, REAL
IM) C; or shorter;

STRUCTURE (REAL RE, IM)
C;
```

2nd possibility:
```
TYPE COMPLEX = STRUC-
TURE (REAL RE, IM)
/* PURE TYPE DECLARA-
TION */;
COMPLEX C
/* (NOT NECESSARILY
DIRECTLY) FOLLOWING
IDENTIFIER DECLARATION */;
```

The declaration of a structure type therefore contains a list of the elements of the structure. The list elements each contain a declaration symbol (e. g. REAL, INTEGER etc.) and an identifier which is used to address the structure element so designated (see below). If the same declaration symbol applies to a consecutive sequence of list elements, it only has to be located with the first list element (see1st possibility in example above).

A structure element is addressed by an element identifier and a structure identifier.

Example: RE OF C resp. IM OF C
or shorter:

RE.C resp. IM.C

An element of a structure, which is itself an element of an array, will be addressed by, e. g.

ELEMENT 3 . ARRAY (I)

An element of an array, which is itself an element of a structure, will be addressed by, e. g.

ELEMENT 2 (I) . STRUCTURE

Examples of structure constants:

1) (5.7, '101'B, 115, 'ABC')

2) Two element structure constant, whose first element is also a structure:
((5.7, 'ABC'), 97)

3) Structure constant, whose second element is an ADDRESS constant:
(5.7, ANTON)

4) Structure constant, whose second element is a one-dimensional three element array constant:
(5.7, (1, 2, 1))

2.2.3. Example of a circular concatenated list

```
(1 : 100) STRUCTURE (REAL CONTENTS, ADDRESS
REFERENCE) LIST
    /* DECLARATION OF THE IDENTIFIER LIST WHICH
       INDICATES AN ARRAY OF STRUCTURE
       ELEMENTS. */;
FOR I TO 99 DO REFERENCE . LIST (I) : = LIST (I + 1);
REFERENCE . LIST (100) : = LIST (1)
    /* OCCUPANCY OF THE STRUCTURE ELEMENT
       REFERENCE IN ALL LIST ELEMENTS WITH THE
       SUBSCRIPTED IDENTIFIER OF EACH OF THE
       NEXT LIST ELEMENTS. */;
ADDRESS POINTER : = LIST (100)
    /* DECLARATION OF THE ADDRESS-VARIABLE
       POINTER WITH INITIALIZATION. */;
```

The result is illustrated by figure 1.

An entry into the list, thus prepared, will be effected by the following two statements:

```
POINTER : = REFERENCE . VALUE POINTER
    /* SET POINTER TO NEXT LIST ELEMENT. */;
CONTENTS . VALUE POINTER : = /* CONTENTS TO BE
    ENTERED */;
```

2.2.4. Conversions between data of different types

Type conversions which are required frequently in expressions, are done automatically. Standard functions are provided for further conversions. When the compiler recognizes that a type conversion is to be made in an expression, a warning message may be output.

2.3. Operators

2.3.1. Classification

Provision has been made for the monadic operators + and - for arithmetic operands, NOT for BINAL operands as well as VALUE for ADDRESS variables and the following dyadic operators:

1) for arithmetic operands:

exponentiation
integer division,
remainder of integer division (modulo),
normal division,
multiplication,
addition,
subtraction
and the conventional six comparisons
($<$, $>$, $\leq$, $\geq$, $=$, $\neq$)

2) for BINAL operands:

conjunction,
disjunction,
equivalence,
antivalence,
shift (direction indicated by the sign of the number of shift-steps), selection of an individual bit and concatenation.

3) for STRING operands:

concatenation
and comparisons;

4) for ADDRESS operands:

comparisons.

2.3.2. Operator Declarations

Operator declarations are provided with the restriction that only operator symbols which already exist can be extended to newly declared types by using such a declaration; the existing operator priorities must not be altered.

2.4. Executable Statements

The following are provided:

- assignments,
- go to statement,
- repetitive statement similar to ALGOL 68,
- switch statement similar to CASE clause in ALGOL 68,
- compound statement,
- conditional statement (IF statement in ALGOL 60),
- interrupt response (ON statement, see 4.3.1.),
- task statements (see 4.2.)
- procedure statement (see 2.1.4.),
- statements for SEMA variables (see 4.4.3.).

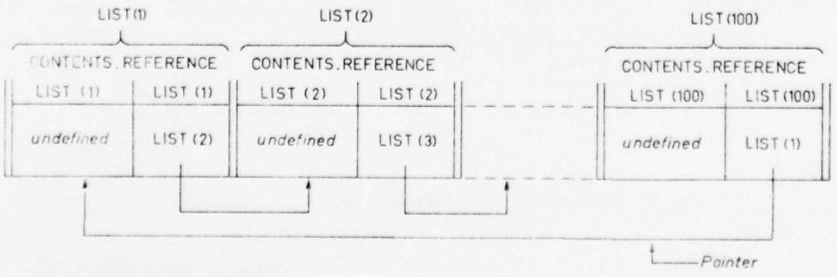


Fig. 1
Example of a circular concatenated list

2.5. Standard Functions and Standard Procedures

Besides the usual mathematical standard functions, standard functions and standard procedures for bit handling are also provided (selection of bit sequences from BINAL data); string handling (selection of characters and character sequences from STRING data), type conversions, input/output (see 5.) and special functions are likewise provided for.

3. The System Division

3.1. Purpose

The system division connects the actual process-program with the environment. It is used to describe the machine configuration on the language level.

It contains parts of a "job control language", it supplies information for the generation of that part of the operating system that is necessary for the special 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 part.

An extensive use of symbolic names in the problem division makes programming easier and improves the readability of the computer program.

By the computer description the compiler is informed about e.g. the type, model and memory size of the machine used.

The configuration description specifies the extend and the structure of the used peripherals. Each external device and process-interface is connected with an identifier. In the same instance the datapath from the central unit to the terminal device is described. By naming only those peripherals that are used by the program, it may be possible to minimize the I/O-package linked to the program.

In the interrupt description the interrupts generated by the system are matched with identifiers, that can be referred to in the problem division. Here also a certain optimisation of the interrupt decoding routines may be performed.

In the flag description status information of the hardware is depicted onto BINAL strings, and can so be tested in the problem part.

3.2. Syntax of the System Division

```
SYSTEM;
MACHINE;
   .
/* COMPUTER DESCRIPTION */
   .
EQUIPMENT;
   .
/* CONFIGURATION DESCRIPTION */
   .
INTERRUPT;
   .
/* INTERRUPT DESCRIPTION */
   .
FLAG;
   .
/* FLAG DESCRIPTION */
   .
```

3.3. Computer Description

The begin of this part is identified by the keyword MACHINE, it contains information about:

a) type (keyword: MODEL)
b) features of the processing unit
c) size of working storage (keyword: SIZE)
d) channels (keyword: CHANNEL)

The information to be supplied in a special case will have to be provided in the corresponding manual.

As one "PEARL" compiler will be written for a whole computer family, it must be informed about the actual level and the features of the implementation, to be able to guarantee an optimal use of the computer by means of a specially optimized object program.

3.4. Configuration Description

3.4.1. Purpose

a) The operating system is informed about the peripheral devices to be used by the program. For the specified environment an optimal version of the operating system may be generated.

b) As the actual equipment with peripheral devices is not specific to the computer but to the whole installation, and can be changed from time to time, the whole datapath from the central unit to the terminal device must be described definitely. This can generally be done by means of an "address tree", the nodes of which are represented each by a control unit with special attributes and interface specifications. (Fig. 2)

c) Identifiers are adjoined to the special datapaths, which are used in the problem division.

3.4.2. Syntax

```
EQUIPMENT;
[STANDARD [; TOTAL]
[; identification] ... ; ]
[SPECIAL [; identification] ... ; ]
```

The metavariable identification is expanded as follows:

```
identification ::= identifier [(index-list)] =
                   [computer-link] {* | ≠} [channel-type
                   [, attribute] ... [(index-list)]]
                   {* | ≠} ... [terminal-type];
```

index-list is a list containing positive integers, similar to that in a loop-specification; an item of the list can be a single number (e.g. 2, 7, 11), a range of numbers (e.g. 3:10, 17:21), or a range of numbers with an increment (e.g. 4(3)16 means the series 4, 7, 10, 13, 16). * or ≠ represent a node in the address tree.

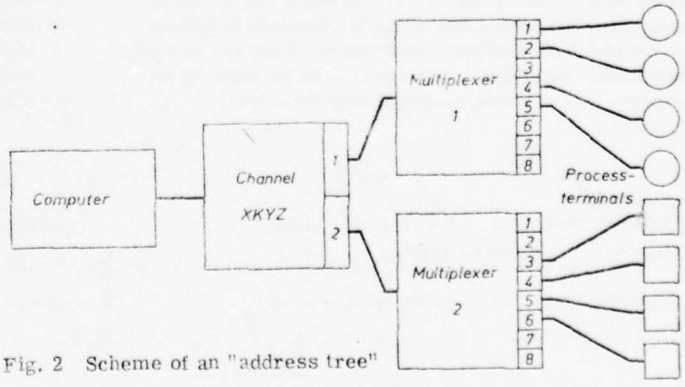


Fig. 2 Scheme of an "address tree"

≠ has an additional function: for two consecutive symbols ≠ that part of the previous datapath description that is enclosed in two symbols ≠ is substituted.

Examples:

1.) DISK = 1*KW, 2*ZWW, FLOAT*STØ5A;

The disk of type STØ5A shall be connected via the floating data-channel ZWW and the 2nd exit of the I/O-processor KW to exit 1 of the computer.

2.) The system of fig. 2 could be described as follows:

TERMINAL(1:4) = XKYZ, 1*MPX1(1,2,4,5)*CIRCLE;
TERMINAL(5:8) = XKYZ, 2*MPX2(3:6)*SQUARE;

3.) Substitution of parts of the datapath description shows:

FIRST = 10 PROCA ≠ CHAN 1*CONTROL 2 ≠ DEVICE 1;
SECOND = 10 PROCB ≠≠ DEVICE 2
/* MEANS: SECOND = 10 PROCB ≠ CHAN 1*
CONTROL 2 ≠ DEVICE 1 */;

3.4.3. Semantic of the Configuration Description

EQUIPMENT introduces the configuration description, STANDARD that of the standard peripherals, SPECIAL that of process oriented peripherals. TOTAL means: all of the standard peripherals connected to the system shall be used.

Standard peripherals have a fixed datapath. They are identified by a name that is defined by the implementation (this "name" may also be an integer); it may be changed in the system division. Both identifiers may be used in the problem division. This is useful, if a program is composed of parts of different origin, or if peripherals are replaced by devices of another type (but with the same function with respect to the program).

identification describes the datapath by a series of type and address items. Details (the keywords and the local syntax of channel-types and attributes) depend on the implementation and can be taken from the manual and the hardware plan of the installation.

The identifier defined in the system division is used in the problem division as an actual parameter of I/O and library procedures and stands for the device address resp. device number. Further parameters are not supplied by the system division.

If a one dimensional array of identifiers is adjoined to a set of identical process terminals, this is described by index-list.

Example:

APC(1:12) = CCTR*3*(1,3,5:7, 12:18)*5*KAD5;
Also arrays of names may be redefined:
HEAT(1:50) = TEMP(20:69);

3.5. Interrupt description

3.5.1. Purpose

The operating system is informed about the interrupts to be used. These are "logical interrupts", which not necessarily have their origin in the hardware and may include additional information about the status change causing the interruption. It shall be possible to generate or simulate an interrupt by the program (see 4.3.2.).

3.5.2. Syntax

INTERRUPT;
[STANDARD[; TOTAL][; interrupt-identifier]...;]
[SPECIAL[; TOTAL][; interrupt-identifier]...;]

The metavariable interrupt-identifier is expanded as follows:

[identifier =]... implementation dependent descriptor;

3.5.3. Semantics

The interrupts listed in the interrupt description can be addressed with the corresponding symbolic names in the problem division. The interrupts not mentioned in the system division are disarmed, disabled or handled by dummy routines. The priorities of the interrupts and the handling routines are prescribed by the order of the interrupt list, if this is allowed by the system.

Interrupts of type STANDARD have predefined names. It is desirable, that interrupts of the same origin should have identical names. The "PEARL"-committee will issue proposals for that.

Interrupts of type SPECIAL are handled over from the operating system to the user program; they are specific for the particular process.

Double definitions are possible.

Example: HEATALARM = PRESSUREALARM = 37;

In many cases the system supplies further information during an interrupt. It can be accessed by the standard procedure STATUS (interrupt-identifier).

3.6. Flag-Description

3.6.1. Flags are status messages of the peripherals or of the computer, that do not cause an interrupt. The operating system is informed, which of them will be used.

3.6.2. Syntax

FLAG[; TOTAL];
[flag-identifier =]... operating system defined name of the flag;

3.6.3. Semantics

flag-identifiers are used like BINAL strings in the problem division, e.g. in conditional statements. Further information may be accessed via a standard procedure STATUS (flag-identifier) in this case, too.

4. TASK-Management

4.1. Problem

Contrary to a sequentially executed program for off-line calculation of results from a set of data, loaded once only, where the chronological order of operations is necessarily obtained from the program and the data -- programs for on-line control and evaluation of data from industrial processes and experiments have an entirely different structure: They consist of many short sections which are, individually speaking, sequentially executed, but which are, regarding the entire process, executed quasi parallel. They are also interrupted by time intervals in which the computer does not have to compute for the process.

This structure is inevitable, because

(a) the computer must be able to react quickly in response to the many external demands which occur statistically in time,

(b) the urgency of the reactions is differentiated so that every reaction can be interrupted, replaced and continued later in favour of another reaction.

The task-management has to enable this quasi-parallel execution and to control its timing.

4.2. The "Task"

4.2.1. Definition

A user-program is executed in sections which are called "tasks". The base of such a task is a statement, or a sequence of statements combined to form a block. Jumps out of this block are not permitted.

The dynamic execution of this sequence of statements, under control of the operating system defines a task. A task can demand, from the operating system, the execution of additional self-contained sequences of statements, whereby new tasks ("subtasks") will be generated.

This generating process may be performed in several stages. Tasks are simultaneously executed, as far as the devices (computer and peripheral units) permit.

4.2.2. Priorities

If several tasks simultaneously demand the use of the same device or unit, the operating system will decide, according to their priority, to which of the competing tasks the unit will be allocated. The user assigns the priorities to the tasks. When not all of the devices required by a task are allocated to it, then the task is placed in the "waiting state". The priority of a task is a natural number. The lower the number, the higher the priority. The operating system selects the task which will be executed first, if two or more have the same priority.

4.2.3. Task-Generation

A task is generated by the statement

ACTIVATE task-name [WITH PRIORITY priority-number] : statement;

where taks-name is the identifier which addresses the task, priority-number the priority of the task and statement the code to be activated.

If WITH PRIORITY priority-number is missing, the new task is assigned the priority of the generating task;

task-name must be declared as TASK.

Activation of a task means to connect the task name with the code, to set the priority and to notify the operating system of the code to be executed. Tasks having the same name must not be activated at the same time.

4.2.4. Task Operations

Other operations which have an effect on a task are:

a) TERMINATE task-name;
b) DELAY task-name;
c) CONTINUE task-name [WITH PRIORITY priority-number];

TERMINATE ends a task and all its activated sub-tasks (non-interruptible operations which have already been started will be finished first).

DELAY transfers a task into the waiting state (but not its subtasks).

CONTINUE cancels this waiting state (i.e. re-activates the task).

Furthermore,

CONTINUE task-name WITH PRIORITY priority-number;

is used to change the priority of the addressed task.

If the addressed task does not exist, i.e. either it has not been activated, or it is already terminated, then the task operations will be interpreted as dummy operations.

In addition to this there is another standard function

PRIORITY (task-name),

which supplies the priority of the task task-name, as a result, if the task exists; otherwise the result will be a negative number.

4.2.5. Time Control of Tasks

In all task operations the time of their execution may be specified as follows:

[AT time 1] [EVERY time-interval [UNTIL time 2]] task-operation;

This type of statement results in the execution of the specified task-operation in intervals of time-interval from time 1 to time 2. If AT time 1 is left out, task-operation will be executed for the first time when this statement is executed. If EVERY time-interval UNTIL time 2 is left out, then a "once only" execution of task-operation follows. In the case that UNTIL time 2 behind EVERY time-interval is missing, then task-operation will be repeated, providing that the task still exists which contains the pertinent statement.

Further activations of a task, caused by a statement which has already been executed, such as EVERY time-interval ACTIVATE task-name; can, nevertheless, be hindered by using another task operation

PREVENT task-name;

PREVENT is ineffective if no further activations are queued for the addressed task.

4.2.6. Task End

A task ceases

a) when the task and all its subtasks have executed their last statement

b) if it is ended by TERMINATE.

If the end of a block is reached, during execution of a task, the execution will be continued only after all activated subtasks in the block are finished.

4.3. Alarms

4.3.1. Reaction to Alarms

Reactions to alarms are specified by the statement

ON alarm : statement;

where alarm is the identifier of a logical interrupt and statement is the code to be executed in response to the interrupt; statement will always be executed with the highest priority. This execution is non-interruptible.

4.3.2. Signal

A logical interrupt interrupt can be generated or simulated in the user program (e. g. for test runs) by the statement:

SIGNAL interrupt;

4.3.3. Disconnection

If several ON statements refer to the same interrupt, then the last one to be executed in the block is valid. When exiting a block, the last valid ON statement in the next outer block is effective. If it should occur that in none of the outer blocks an ON statement is executed, then there is no further response to this interrupt.

An interrupt, at a distance as far as possible from the central unit permitted by the implementation concerned, can be switched off by using

DISABLE alarm;

The interrupt can be switched on again by

ENABLE alarm;

4.4. Semaphore

4.4.1. Problem

Since the timing sequence of operations in two tasks running parallel with each other cannot, in many cases, be completely arbitrary, means must be provided whereby the desired sequence of such operations can be attained. For instance – before the output of an array by a task, this array has to be filled in by another task. If this sequence is accidentally reversed then the results are senseless. For the synchronization of tasks running parallel with each other, semaphore variables and non-interruptible semaphore operations which have an effect on the semaphore variables are used.

4.4.2. Features of the Semaphore Variables

Semaphore variables are initialized at the same time as their declarations. Further access to a semaphore variable is only possible via a semaphore operation.

4.4.3. Operations

Semaphore operations are non-interruptible. The following operations are available:

REQUEST sema
RELEASE sema

The operation REQUEST decrements the value of the semaphore variable sema by one, providing that the result is not negative. If the result were negative, the task which is trying to decrement the semaphore variable will be placed into the waiting state until the semaphore variable can be decremented to a non-negative result.

RELEASE increments the value of the semaphore variable sema by one and starts the tasks, in order of priority, which had previously been stopped by REQUEST operations.

The operators RELEASE and REQUEST may operate on lists of semaphore variables. Consequently,

REQUEST sema [, sema] ... ;

checks whether all semaphore variables in the list can be decremented to a non-negative value. If this is impossible, then none will be drecremented and the operation will be stopped until the possibility arises.

4.5. Working Storage Allocation

4.5.1. Principle

The entire code of a PEARL-program will, in many cases be located in backing storage and not in working storage. Consequently the code needed for a task has to be loaded, by the operating system, from the backing storage to the working storage. The operating system determines from the activation statement for this task, that additional code is required as well as the identification of this code. The activation statement must be executed early enough so that the loading procedure is finished in time. Untimely execution of the task can be prevented by using SEMA variables.

4.5.2. Code of a Task

The code required for a task consists of all instructions which may be performed and all data which will be formally used therein. In particular, all procedures which will be called in the task (possibly via several stages) belong to the code. The requirements of a sub-task, e. g. its instruction sequence, do not form part of the code of the task activating it.

As in RTL [4], procedures and data will not be loaded together with the block containing their declaration, but together with the instruction sequences in which they will be (formally) called or used.

4.5.3. Block Residence

It is possible for the PEARL-programmer to load procedures, i. e. their instructions and data, together with the code of a task not containing a call for them. This is attained by the statement

RESIDENT procedure-call [, procedure-call] ... ;

The effect as regards loading is the same as that of calls located in the same position. Hence RESIDENT applies only the block in question.

4.5.4. Displacement

For loading, the section of working storage which is not occupied with instructions or data of an existing task will be used first of all. If this space is no longer available, then the code of a sufficient number of tasks which are either in the waiting state or have a lower priority than the task to be newly activated, will have to be displaced from the working storage. The data to be displaced must not be overwritten and must, therefore, be previously transferred into the backing storage. If there are no tasks with lower priority than the one be newly activated, then it will be placed in the waiting state.

4.5.5. Reload

Generally speaking, tasks to be newly activated often require data and procedures which have already been called or used by tasks which chronologically precede them, and which are consequently still located in working storage. This raises the problem for the operating system of not only determining the additional code required, but also of linking this code to sections which are already in the working storage.

Whether this is possible, and how to achieve it, depends largely on the computer system in use and its operating system. Since working storage allocation is controlled by priorities, it may be necessary to restrict the priorities of sub-tasks in order to obtain effective implementation.

5. Input/Output

5.1. Basic Principles

If one attempts to handle all known I/O-facilities from a uniform point of view, even in commercial or scientific data-processing-applications difficulties arise; e.g. external storage devices with either random or sequential access should be addressed in the same manner. Apart from that there are functionally very different I/O-devices (compare a teletypewriter with a CRT) that shall be handled with procedures that are as similar to each other as possible.

The additional peripheral devices of a computer that is used for data acquisition and control of industrial processes and scientific experiments enlarge the difficulties of a standardized I/O-description. In this section it is tried to develop a syntactically uniform description of all I/O-facilities.

Primarily the uniformity of the description is achieved by the formulation of procedures for all I/O-functions. In the formal handling this corresponds to the possibilities that are provided in ALGOL 60 and 68.

In FORTRAN and PL/1 I/O-transfers are formulated as special statements, but also in these languages they are implemented as subroutines.

In PEARL this procedure concept yields another advantage: we have the possibility to easily activate I/O-operations in a sequential and in a parallel manner:

ACTIVATE: OUTPUT 1 (CA, RAM, BA); describes output done in parallel with the execution of the following statements.

The procedure call "OUTPUT 2 (TOM, DOOLEY);" without the prefixed ACTIVATE starts the I/O-procedure as a subroutine. This means: the running task executes the succeeding-statement after the termination of the I/O-operation.

The description of I/O procedures in PEARL is characterized by 5 descriptors:

direction, mode, external-terminal, internal-terminal, option. Direction (IN or OUT only) and mode are combined to one word (the name of the procedure). The following three descriptors are parameters of this procedure.

external-terminal is an identifier for a device or a file. internal-terminal defines the area in working storage where data is to be in- or outputted. Syntactically it is represented by a (if necessary parenthesized) list of identifiers. option describes control information which is probably necessary for special modes.

Six mode-descriptors seem to be sufficient:

PRIMITIVE
BINARY
CHARACTER
GRAPHIC
CALIBRATED
CONTROL

Suitable abbreviations of these keywords will be defined later by the PEARL-committee.

5.2. Syntax, Semantics and Function of the I/O-Procedures

5.2.1. PRIMITIVE Input/Output

Syntax:

{INPRIMITIVE / OUTPRIMITIVE} (external-terminal, internal-terminal [, control-information]);

control-information is a list of variable identifiers and/or constants containing control information for the data transfer.

Semantics:

The procedures INPRIMITIVE and OUTPRIMITIVE are used to transfer data from a terminal device to working storage and vice versa; information for datapath- and device-control must be provided explicitly by the programmer. This can be done by the control-information included in the same procedure call or by a previous output of control information only.

Examples:

```
1) INPRIMITIVE (DEVICE1, DATA, '101101101111'B)
   /* TRANSFER FROM DEVICE1 TO THE VARIABLE 'DATA' IS
   CONTROLLED BY THE BINAL STRING OF PARAMETER 3 */;

2) X := '00000100'B;
   AB := '0110'B;
   OUTPRIMITIVE (PROCON3, X, AB)
   /* THE FOLLOWING INFORMATION IS TRANSFERRED TO
   PROCESSCONTROLUNIT 3 :
   AB IS INTERPRETED IN THAT WAY : 01 SELECT THE 2ND OF 4 STEP
                                      MOTORS,
                                   10 TURN CLOCKWISE,
AND X : TRANSFER THE VALUE 4 TO THE MOTOR'S STEP COUNTER */;
```

Range of application:

PRIMITIVE Input/Output permits the possibility to handle machine code on language level. Primarily it is necessary if none or only rudimentary routines for handling an addressed peripheral and its datapaths exists (e.g. new built or modified hardware). All other I/O-functions can be expressed by suitable combinations of PRIMITIVE I/O.

5.2.2. BINARY Input/Output

Syntax:

{INBINARY / OUTBINARY} (external-terminal, internal-terminal);

Semantics:

The procedure OUTBINARY transfers data from working storage to a terminal device while control of datapath and device functions is entirely performed by the operating system. The transferred information is not changed, in particular it is not translated into any other code.

In the same manner the procedure INBINARY has as its result an automatic transfer without data transformation.

Range of application:

The procedures for binary I/O can e. g. be used for intermediate storage of data on magnetomotoric storage devices (similar to the use of "unformatted I/O" in FORTRAN). Reading from and writing onto process peripherals is also done with these procedures when there is a standard mode of transfer and no calibration of data is necessary.

Example:

```
INBINARY (TEMP(1:4), KELVIN(5:8))
/* INPUT FROM THE FOUR THERMOCOUPLES 'TEMP (1)' TO 'TEMP (4)'
IS TRANSFERRED UNCHANGED FROM THE ADC TO THE ELEMENTS 5
TO 8 OF ARRAY 'KELVIN' */;
```

5.2.3. Input and Output of Characters

Syntax:

```
| INCHARACTER  |
|OUTCHARACTER  | (external-terminal, internal-terminal,
                 {FORMAT (string), PICTURE (string), SPECIALLAYOUT(...)});
```

The third parameter of the CHARACTER-procedures is a *format-describing procedure*, the parameter of which is a character string.

Semantics:

Together with the data transfer a translation (code-transformation) is performed character by character between the internal machine representation of the data (which is defined by the special machine and by the data type) and the external representation (defined by the device, and occasionally by the programmer, included in the format description). Moreover the layout of the data on the external medium is described by the FORMAT-(or PICTURE-) procedure.

It shall be possible to specify special layout procedures instead of the standard ones (e. g. procedures that include interpretation of commands entered on a manual input device).

The external-terminal is regarded as a book as in ALGOL 68 [13]; (also refer to the file description in 2.2.1.5.). It is suggested that the parameter-string of the FORMAT-procedure should be identical or very similar to that of PL/1 [8] (Probably including a code-definition facility). This features should be supplemented by a new-defined PICTURE-procedure which corresponds to a layout-description which is very simple to use.

Yet, it is discussable that, instead of the aforegoing points, the complete concept of any other programming language, which has advanced text editing features, may be utilized.

5.2.3.1. The PICTURE-procedure

The single parameter of PICTURE is represented by a character-string that depicts the layout on the external medium (e. g. page of a line printer) both in length and structure. String constants are inserted into the string as they shall be outputted, fields that are to be filled with variables are reserved by space imprinted with the wanted layout structure.

Example:

The variables I (integer, value 23), B (real array with two elements, values 17.1 and $3.1*10^{12}$) and TXT (string, value 'PEARL') shall appear on an external device like teletype-writer or CRT, connected to "RESULT-LIST", showing this layout:

```
RESULT
TEST 23                    B1 = 17.1        B2 = 3.1E12
----------THIS WAS A PEARL-PROGRAM----------
```

The output procedure is called in this way:

```
OUTCHARACTER (RESULTLIST, (I, B, TXT), PICTURE
('RESULT
 TEST # #                  B1 =## $# B2 = # $ # $ # #
----------THIS WAS A eeeee-PROGRAM----------'));
```

The variables I, B, and TXT are inserted into the fields structured with the symbols ≠, $ and e. Their special meaning is:

≠ : replace with a number
$: replace with a special symbol (also E in exponent notation)
e : replace with an alphanumerical symbol

If a string-constant contains these symbols, they must be enclosed in double quotes. Perhaps further special characters will have to be used for picture specifications (e. g. for suppression of leading zeroes).

The purpose of this modification of conventional PICTURE-procedures is to enable the user to describe the exact layout of the text without any shifting caused by delimiters.

5.2.3.2. Range of application

CHARACTER I/O shall be used everywhere where character strings (text or numbers) are in-/outputted. The I/O-devices will in most cases be dedicated to man-machine communication. This method may also be used to handle process I/O if the peripherals use alphanumeric coded data representation.

Example:

```
INCHARACTER (ADC 1, VAR, PICTURE ('####␣␣'))
/* THE FIRST FOUR OF THE SIX DECIMAL POSITIONS OF ADC 1
SHALL BE DECODED AND STORED INTO VAR */;
```

CHARACTER I/O is also useful for tape- and disk-like devices if they are e.g. used as buffer storage for slow peripherals.

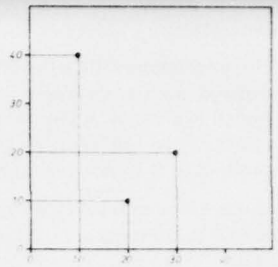


Fig. 3

Example of a RANDOM point display

5.2.4. Graphic Input/Output

By means of this procedure all I/O related to graphs and pictures can be described. It handles the typical graphic devices like plotters or CRT-displays, but also the "drawing" of a curve by means of a typewriter with points represented by some letter (e.g. 'x') will be done using the GRAPHIC mode.

2) SPECTR := (40.0, 35.0, 25.0, 15.0, 13.0, 17.5, 25.0, 31.5, 37.0, 39.0, 40.0, 39.5, 37.5, 33.0, 22.5, 10.0, 5.5, 3.0, 2.0);
OUTGRAPHIC (DISPLAY, SPECTR, INCREMENT (2.5));

5.2.4.1. The output Procedure OUTGRAPHIC

Syntax:

OUTGRAPHIC (external-terminal, internal-terminal, layout-procedure (par, ...));

The user of PEARL will be supplied with a number of standard layout-procedures for graphic editing. He may as well write his own routines for that purpose.

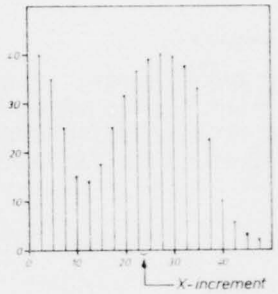


Fig. 4

Example of an INCREMENTal display

Standard layout-procedures:

no.:	internal data:	procedure name:	parameters:	graphic picture:
1)	$\vec{x}$, $\vec{y}$ [, $\overrightarrow{intens.}$]	RANDOM	intensity, scale	see example 1)
2)	$\vec{y}$ [, $\overrightarrow{intens.}$]	INCREMENT	intensity, y-scale, x-increment	see example 2)
3)	$\vec{x}$, $\vec{y}$, $\overrightarrow{intens.}$	LINE	scale	see example 3)
4)	$\vec{x}$, $\vec{y}$ [or $\vec{y}$]	INTERPOL	intensity scale, mode of interpolatn.	see figure 6
5)	$\vec{x}$, $\vec{y}$, $\vec{r}$, $\vec{\varphi}_1$, $\vec{\varphi}_2$	CIRCLE	scale, intensity	

3) STAR := ((10, 10, '0'B),
(20, 20, '1'B),
(30, 10, '1'B),
(20, 30, '0'B),
(20, 20, '1'B));
OUTGRAPHIC (DISPLAY, STAR, LINE);

The basic method of graphic layout is the RANDOM point plot (1), points on the image correspond to pairs of values (coordinates) in a data array. The image may be modified by specification of scale factors and of the brightness of the points (if possible). With INCREMENT point plot the facility of many graphic devices to generate x-coordinates automatically (1, 2, ... n) is described (2). The programmer must provide the y-data only.

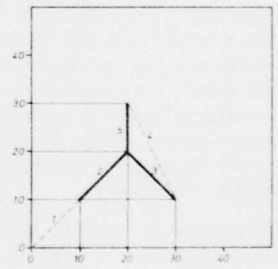


Fig. 5

Example of a vector (LINE) mode display

With the LINE ("vector") mode two consecutive points are connected with a (bright or dark) straight line. Only the specification of the brightness and one terminal point is needed as the drawing beam starts at the previously reached position.

Special layout functions for INTERPOLation (4) and for the drawing of arcs of CIRCLEs (5) may be useful.

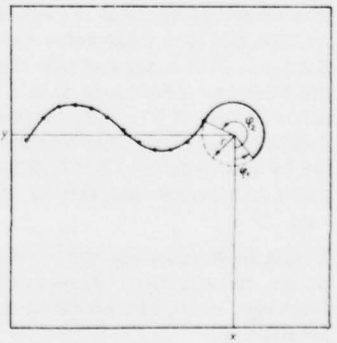


Fig. 6

Example of an INTERPOLated curve and an arc of a CIRCLE

Examples:

1) X(1) := 10; X(2) := 20; X(3) := 30; Y(1) := 40; Y(2) := 10; Y(3) := 20;
OUTGRAPHIC (DISPLAY, (X, Y), RANDOM);

5.2.4.2. The Input Procedure INGRAPHIC

Syntax:

INGRAPHIC (external-terminal, internal-terminal, function (par, ...));

Semantics:

external-terminal is a graphic input device like a lightpen or a joystick that has been described in the system-division of the program. Usually internal-terminal is a pair of variables (x, y), and there is no 3rd parameter. The input device supplies coordinates which are stored in x and y. INGRAPHIC allows the identification of certain points on the image on a graphic output device. According to that identification further actions can be taken in the program.

5.2.5. Calibrated Input/Output

These procedures are useful for transferring analog information supplied by a process input device; during output the user is able to specify an angle e.g. in units of degres instead of a number of steps a motor has to turn.

5.2.5.1. The Input Procedure INCALIBRATED

Syntax

INCALIBRATED (external-terminal, internal-terminal, calibration-function (par, ...));

Semantics:

The external device supplies one or more analog test values that are stored in working storage in a digitalized and calibrated form (transformed into physical units). This is accomplished by a calibration-function that is part of the system library, the parameters of which can be supplied by the programmer. E.g. one can use a polynomial to calibrate the input values delivered by a thermocouple. When the element is replaced this can be taken into account merely by changing the coefficients of the polynomial (= parameters of the calibration-function).

5.2.5.2. Output using OUTCALIBRATED

Syntax:

OUTCALIBRATED (external-terminal, internal-terminal, calibration-function (par, ...));

Semantics:

internal-terminal contains variables that specify the amount in physical units by which the position of a terminal device (e.g. a valve using servo motors) has to be changed.

Example:

```
OUTCALIBRATED (MOTOR1, FLOW, CALIB1(A1, A2, A3))
/* THE FLOW THROUGH A PIPELINE IS REDUCED TO A CERTAIN VALUE;
THE RELATION BETWEEN THE POSITION OF THE VALVE AND THE VALUE
OF THE FLOW IS NONLINEAR AND DESCRIBED BY FUNCTION 'CALIB1' */;
```

Comment: The calibration-procedures must be predefined by the programming system. A linear calibration function, a polynomial and a nonlinear calibration using several lists seem to be sufficient for the most applications.

5.2.6. Input and Output of Control Information

Syntax:

{OUTCONTROL / INCONTROL} (external-terminal, internal-terminal);

(In the case of OUTCONTROL the internal-terminal may also be a constant – generally BINAL)

Semantics:

INCONTROL stores the status information (as far as it is relevant to the system) of the addressed peripheral *into the named variables*. Using OUTCONTROL control commands can be sent to peripheral devices.

Range of application:

INCONTROL mainly will be used to check functions of process peripherals; this procedure is a means of machine-oriented programming. In a similar manner OUTCONTROL is used if one wants to deal with details of the I/O-hardware (as it can be done using assembler language).

The features of CONTROL I/O are similar to that of PRIMITIVE I/O; but here only control information – no data – is exchanged.

Example:

```
ON INTERRUPT SENSOR1 BEGIN
        INCONTROL (SENSOR1, X);
        IF X = '00110010'B THEN GO TO ALARM FI
        END;
```

5.3. Auxiliary Procedures for Input/Output

To accomplish complex control functions in the I/O-management of the system a number of standard procedures are available.

These are procedures for OPEN-ing and CLOS-ing I/O which can also act on files, for reserving a device for exclusive use (LOCK) and for cancelling the reservation (UNLOCK), for RESET-ting a device and for SET-ting it to a specified state and for STOP-ping I/O abruptedly. Functions to CREATE, to REIDENTify and to SCRATCH (i.e. to cancel the reservation) files are also necessary. There must be also a number of formatcontrolling procedures (to be used as subparameters of FORMAT or layout-procedures or as stand-alone calls), e.g. BACKSPACE, SKIP, NEWLINE, and NEWPAGE.

6. Acknowledgements

This proposal of a process- and experiment-oriented programming language was composed by a working group which representatives of the following firms and institutes belong to:

1. Hahn-Meitner-Institut für Kernphysik, Berlin
2. Physikalisches Institut III der Universität Erlangen
3. Strahlenzentrum der Universität Gießen
4. Zentralinstitut für angewandte Mathematik der Kernforschungsanlage Jülich
5. Gesellschaft für Kernforschung Karlsruhe
6. Siemens AG Karlsruhe
7. AEG-Telefunken Konstanz
8. BBC Mannheim
9. Physikalisches Institut III der Universität Marburg
10. Math. Institut der Technischen Hochschule München

Here we want to thank you who have made this work possible by sending delegates; especially also Mr. Heller (BASF, Ludwigshafen), the chairman of the VDI subcommittee "Programmiertechnik" who has maintained the connection to our working group. We also want to thank Prof. Dr. N. Fiebiger from the Physics Institute III of the University of Erlangen for his interest and helpful support, the Bundesministerium für Bildung und Wissenschaft and the "Studiengruppe Nuklearelektronik" for financial support. Last but not least Mr. Kessel and his co-workers from the Institut für Kernphysik der Universität Frankfurt have helped us very much by providing for rooms and organizing the sessions.

Beside the authors the following participiants have contributed to this paper:

M. Degenhardt (HMI Berlin), H. Homrighausen (KfA Jülich),

G. Koch (BBC Mannheim),

K. Kreuter (Siemens AG Karlsruhe),

W. Rüb (TH München),

W. Schöfer (AEG-Telefunken Konstanz),

D. Wiegandt (Univ. Marburg, now at Cern Genf),

K. Wölcken (Universität Gießen).

If you want copies, or if you have any comments or questions, please write to Mr. P. Elzer, Tandemlabor, Physikalisches Institut, University D-852 Erlangen, Erwin-Rommel-Str. 1.

7. Literature

[1] Opler: Requirement for Real-Time-Languages; Comm. ACM 9 (1966), 196

[2] Zemanek: Rolle und Bedeutung formeller Sprachen; E u. M 83 (1966), 463

[3] Frost: Fortran for Process Control; Instrumentation Technology 16 (1969), 4

[4] "RTL" A language for Real-time-systems; The Computer Bulletin, Dez. 1967, S. 202-212

[5] Boulton, Reid, Pierce: A process control language; IEEE Transactions on computers C-18 (1969), 1049

[6] INDAC-8 software: Digital Equipment Corporation; Maynard, Mass.

[7] G. Müller: Die Verwendung einer problemorientierten Sprache für Prozeßrechner, Aufbau und Funktionsweise des zugehörigen Compilers; Vortrag auf dem "Jahreskolloquium zur Rechentechnik", Febr. 1970, TU Braunschweig

[8] PL/1-language-specifications, IBM C28/65-71

[9] Dijkstra: Cooperating Sequential Processes in: Genuys (Editor); Programming languages, London 1968

[10] Aus: PL/1 – Sprachspezifikationen: IBM Form 79879-1

[11] Wijngarden, Mailloux, Peck, Koster: Report on the Algorithmic Language ALGOL 68; Numer. Math. 14 (1969), 79-218

[12] Beech et al.: Concrete Syntax of PL/1; IBM United-Kingdom-Lab. TN 3001

[13] Naur et al.: Revised Report on the Algorithmic Language ALGOL 60; Num. Math. 4 (1963), 420-453

[14] V. Haase: EXOS – Entwurf einer experimentorientierten Programmiersprache; GfK-Karlsruhe, externer Bericht, August 1967

[15] P. Elzer: Möglichkeiten zur Entwicklung einer Programmiersprache für kernphysikalische Experimente; Arbeitspapier Phys. Inst. Erlg.; Juli 1968

[16] Berger, Seibt, Strunz: Bibliographie... zum Thema Programmiersprachen, elektron. datenverarb. 11 (1969) Heft 5 und 7

[17] IEEE-Transactions: Industrial & electronics & control instrumentation 15 (1968) No 2 (Sonderheft über Prozeßkontrollsprachen)

[18] Workshop on Standardization of Industrial Computer Languages (Minutes, part 1) Purdue University, Lafayette, Indiana, Febr. 1969

[19] Preliminary Glossary, workshop on standardization of Industrial Comp. Lang. (Minutes, part 2) Purdue University, Oct. 1969

[20] Workshop on Standardization... (Minutes, part 3) Purdue University, March 1970

[21] IFIP Fachwörterbuch der Informationsverarbeitung: Amsterdam 1968

INDUSTRIAL PROGRAMMING LANGUAGE : L A I

(LANGAGE POUR LA PROGRAMMATION DES APPLICATIONS INDUSTRIELLES)

C O N T E N T S

1.- INTRODUCTION

2.- DATA TYPES

2.1.- Arithmetic data
2.2.- String data

3.- DATA AGREGATES

4.- OVERLAY

5.- EXPRESSIONS

6.- INSTRUCTIONS

7.- PROGRAM STRUCTURE

8.- LANGUAGE-SUPERVISOR RELATIONS

8.1.- Events - semaphores
8.2.- Task management
8.3.- Interrupt control
8.4.- Input - output

9.- COMPILATION METHOD

PRECEDING PAGE BLANK-NOT FILMED

L A I

1.- INTRODUCTION -

The LAI language has been developed as the result of the search for new means of programming industrial applications so as to diminish the overall cost.

The work has been financed by an assembly of users (DGRST) with Government funds.

The resulting specifications summarized below, are the work of six people who used the Purdue Workshop LTPL Committee specifications as a guide line and who evaluated a number of existing languages.

PL/1 or Fortran programers must find it particularly easy to learn, but it is not thought that a subset of PL/1 would meet all the functional requirements. Therefore, the language strives, where possible, to resemble PL/1 and/or Fortran.

LAI's principal caracteristics are :

- machine-independance, allowing the re-use of programs, changing computers, package-implementation.
- real-time language. The language is coupled to a standard supervisor, written in the language and includes the concepts of events, semaphores, tasks, interrupt-handling so as to solve specific real-time industrial applications problems.
- fixed-point and floating-point arithmetic.
- bit and bit-string handling.
- data packing and unpacking.
- formalization of industrial input-output.

2.- DATA TYPES -

2.1.- Arithmetic data -

- Integer variables. Attribute : maximum absolute value
- Integer constants. Attribute : none
- Unsigned integer variables. Attribute : maximum absolute value
- Unsigned integer constants. Attribute : none

- Fixed-point variables. Attributes : maximum absolute value and precision
- Fixed-point constants. Attribute : precision
- Real variables. Attribute : number of decimal digits for the representation of the mantissa
- Real constants . Attribute: number of decimal digits for the representation of the mantissa.

2.2.- String data -

- Bit-string variables. Attribute : length of the string number of bits
- Bit-string constant. External representation :binary, octal or hexadecimal
- Caracter-string variables. Attributes : string-length: :and associated code
- Caracter-string constants. Attribute : associated code
- Sub-strings are declarable (static attributes)
- PART pseudo-variable : sub-string definition with dynamic attributes.

3.- DATA AGGREGATES -

- Arrays : aggregate of data of the same type with the *same attributes*.
- Structures : aggregate of data of anay type and attributes. All data (including eventually arrays) declared in a structure are packed in memory. Structures will often be used in structure arrays.

4.- OVERLAY -

The overlay declaration allows the same memory space to be affected to several data or data aggregates, whose attributes may be different.

5.- EXPRESSIONS -

The operands may be integer, unsigned integer, fixed-point, real, bit-string or single caracter. In the case of bit strings and single caracters, a conversion to unsigned integer is implicit. All classic arithmetic operations are allowed : addition, subtraction, multiplication, division, exponentiation.

Conversions between arithmetic types are implicit in the following order of increasing priority : unsigned integer, integer, fixed-point, real. Explicit conversion functions

.../...

- Logical or bit-string expressions

  Boolean expressions are a particular case of logical expressions. The operators are : negation, intersection, union, disjunction. Comparisons give boolean or logical primaries.

6.- INSTRUCTIONS -

- Assignment (with implicit conversion)
- GO TO
- Computed GO TO
- Subprogram call : subprogram name followed by parameters and possible return addresses between parentheses
- RETURN : dynamic (run-time) end of a subprogram, an interrupt procedure, a function or a task
- Conditional instruction : it can have two forms :

  Form 1 :
  IF logical expression THEN unconditional instruction ELSE unconditional instruction

  Form 2 :
  IF logical expression THEN unconditional instruction

Any other instruction is of unconditional type (e.g.a. compound instruction).

- Iteration instruction : its form is :

```
DO variable name - initial value UNTIL maximum STEP value ;
  ¦ body of the iteration
  ¦
ENDDO ;
```

- Compound instruction : it is made up of a number of instructions enclosed between DO and ENDDO ;
- Supervisor instructions : they are detailed in paragraph 8

7.- PROGRAM STRUCTURE -

A program is made up of a number of articles. There can be the following types of articles :

- global data article : this article contains only the declarations of data common to several modules (a module is a separately compilable unit). Any other article referencing these global data must declare them with the EXTERNAL attribute.
- subprogram article : it is the classic notion of subprogram. It can have the attribute REENTRANT. The arguments are passed by value (default option) or by address. The subprogram may return values to the invoking program.

.../...

Several return addresses may be specified, the RETURN, end of the subprogram, being followed by a computed value which enables the selection of the return address in the list.

The subprogram's formal parameters have a presentation which is identical to declarations.

- function article : it has a form identical to the subprogram but with an attribute which gives the result's data type.
- interrupt procedure article : each interrupt procedure is attached to a hardware interrupt by an ON statement. The interrupt procedures have no parameters and cannot contain any supervisor instructions which could place them in a suspended state. They may create tasks and activate events.
- process article : the process is that program unit whose execution (including the subprograms it invokes) will constitute a task. It is managed by the supervisor. The processes cannot be reentrant, i.e, at any time, there can only be one task attached to each process. There is a waiting list attached to each process for storing task creation requests when a task already exists.

  A static priority level is associated to each process ; this will be the priority of the task created unless a dynamic priority is specified at the time of creation.

  A process may have parameters, passed by value or by address.

  One process will be qualified for general initialisation by use of the key-word START.

8.- LANGUAGE-SUPERVISOR RELATIONS -

8.1.- Events-semaphores -

The supervisor manages the system resources by the use of operators applied to two particular data types in the limits specified by the programer.

- Events
  These are booleans managed by the supervisor. EVENT type variables may have one of two values, SET or RESET, controlled by the orders ACTIVATE and ERASE.

  The instruction WAIT "event expression" allows the user to specify the conditions under which the waiting task can again be eligible. The event expression can contain event type variables and times, connected by the logical operators AND, OR.

.../...

- Semaphores

The semaphore variables and their associated operators RESERVE, FREE allow the user to program the management of resources common to several tasks (data-access, programs, peripherals...). The execution of these operators by the supervisor is uninterruptible.

A counter is associated to each semaphore ; its initial value is given by the declaration. The counter is incremented by a value n by the operator FREE with the parameter n and decremented by a value m by the operator RESERVE with the parameter m. The decrementation takes place only if the counter's value after decrementation is not negative ; otherwise, the counter's value is unchanged and the invoking task is suspended.

8.2.- Task management -

A task is created by invoking the corresponding process. If a task was already created for that process, the call is placed in the attached waiting list.

The execution of the RETURN instruction in a process ends the associated task.

A task may be created either by another task or during the execution of an interrupt procedure.

The supervisor's scheduler gives CPU control to the highest priority task immediately after a task creation or at the end of an interrupt procedure. The scheduler also gives control to the highest priority eligible task every time a supervisor operator is executed which can change the eligibility of the current tasks.

A task may be suspended by the instructions WAIT, RESERVE, INPUT, OUTPUT and after creation of a higher priority task or if a higher priority task becomes eligible.

The instruction STOP suspends unconditionally the task associated to the process whose name is given. The instruction GO renders it eligible again.

Any task may be interrupted by an interrupt ; it will regain control after the interrupt procedure has been executed, unless a higher priority task has been created or rendered eligible in the procedure.

8.3.- Interrupt control -

The interrupt control instructions allow the connection of an interrupt procedure to an interrupt level, to enable, disable, arm, disarm any interrupt level.

The instruction WAKE (delayed software interrupt) will cause the execution of an interrupt procedure after a given time has elapsed.

8.4.- Input - output -

Standard input - output (computing - center peripherals) are separated from industrial input-output peripherals.

- Standard input-output

The I/O instructions reference formats not unlike Fortran's and can be executed with or without wait. If executed without wait, an event can be associated to the I/O ; this will be activated by the supervisor at the end of the I/O.

- Industrial input-output

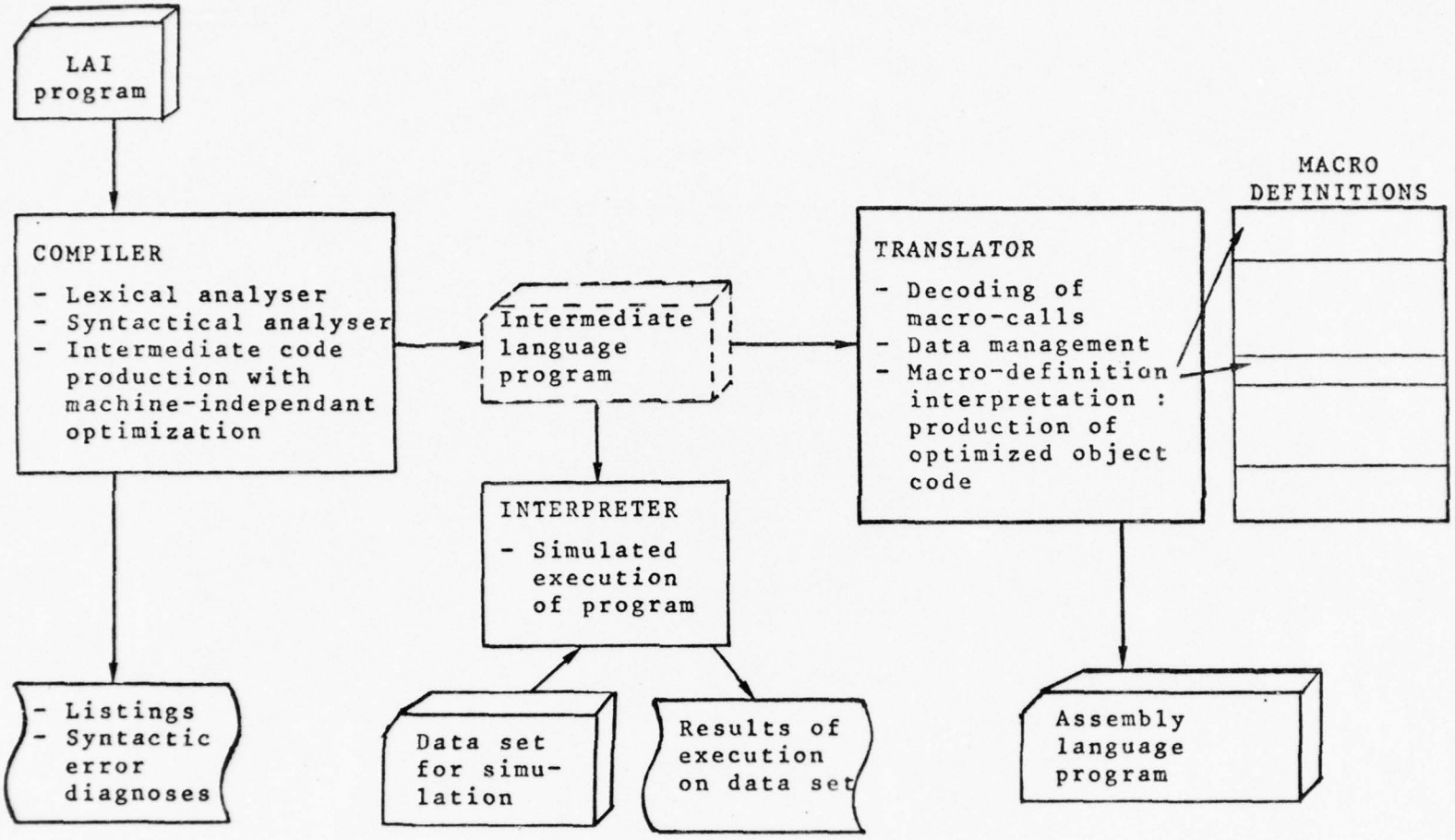
An attempt of formalization of industrial I/O has been made. The I/O by canal and those by programed link are distinguished. In both cases, all of the parameters of the I/O instruction and the associated peripheral declaration are programed in a machine-independant way.

This is a point which will be completely determined only after the first implementations.

9.- COMPILATION METHOD -

The method of compilation is given on the following figure. The compiler has been evaluated at about 120 k bytes.

.../...



L A I COMPILATION

- The COMPILER and TRANSLATOR blocks are machine-independant
- The macro-definitions must be renewed for every object-machine
- The interpreter allows a good debugging
- The operator does not need to know the intermediate language

PRECEDING PAGE BLANK-NOT FILMED

LTPL EUROPEAN GROUP
LTPLE / 016

Categories : I , D
Update none
Obsolete none
5 pp

G. LOUIT
STERIA-FRANCE
15 December 71

ELEMENTARY DESCRIPTION OF THE FEATURES OF THE PROCOL LANGUAGE

Introduction

The purpose of this note is to present the main features of the PROCOL Language in the same manner as it was done in Appendix A of the minute of the 2nd meeting in Paris (March 16-17 - 1971).

PROCOL has been developed on the initiative of the Automation Commitee of the D.G.R.S.T. * and this was decided after a census of the fonctional requirements has been done.
It appears like a general language for Real Time applications.

A first implementation of PROCOL is being done on the T 2000 of LA TELEMECANIQUE and is composed of four parts which will not be described in details here :

1	An executive system	PREX
2	A compiler	PROC
3	A system generator	PROG
4	A loader	PROL

1.1 - Basic Language Structure

Language Character set

Set of ASCII characters :

A, B, ... Z

0, 1, ... 9

[ ] \ ↑ ← SP ! " # $ % £ ' ( ) * + - , . / : ;

< = > ? @

Delimiters

Operator

Parentheses

Keywords

Identifier

A letter followed by up to 5 letters or digits only.

Keywords

reserved

Comments - enclosed by ':' and ';'

./.

1.2 - Basic Programme Structure

Program (programmed application)

A programmed application is a collection of elements ; an element may be :

- Common
- External sub-programs (re-entrant)
- Task (composed of the task itself and eventually of internal sub-programs)

Simple statement

N° Compound statement

N° Bloc Structure

2.1 - DATA ORGANISATION

a. Scalar data
b. Arrays
c. Structures (packed information on one word)
d. Arrays of Structures

2.2 - NAMING

a. Simple name
b. Subscripted name
c. Qualified name
d. Subscripted qualified name.

./.

2.3 - DATA TYPES

a/ Problem data

Integer with sign (one word)
Real with sign (two words)
Binary (may be scaled by a scale factor)
Short integer (T2000)

Structure (data whose length is one word and which can be sliced into parts : items, there items may overlap each other within the structure).
Items corresponding to such structures.
Arrays of each mode in one, two or three dimensions.

b/ Program control data

- Common declarations which include { DATA declaration / I/O declaration / FORMAT declaration }

- Sub programs declaration (External subroutine or Functions)

- Task declaration { SUBROUTINE (or FUNCTION) / PROCEDURE / ----- / MAIN }

2.4 - EXECUTABLE STATEMENTS

Assignment statement
GO TO statement
Switch statement
DO statement
Conditional statement
Subroutine statement
Function statement

./.

RETURN statement
Code insertion statement
Task handling statements
Timers handling statements
Pulses counter handling statements
Interrups and Events handling statements
Semaphores handling statements

3.1 - SCALAR EXPRESSIONS

a/ Arithmetic Operations
addition
subtraction
multiplication
normal division
exponentiation

b/ Bit string opérations
conjunction
union
disjunction

c/ Comparison operations
= ≠ (Different) <= < > >=

d/ Address Operands
- none -

e/ Data conversion
implicit

4.1 - DATA DESCRIPTION-DECLARATION

The declarations are explicit.
A declaration is static and its range is the 'place' where it is made where 'place' may means : task, sub-programms, functions and subroutines. Data in common has a global scope.

LTPL EUROPEAN GROUP
LTPLE/017

Category : I, D4
Update none
Obsolete none
3PP

G. LOUIT
STERIA-FRANCE

22 Décembre 71

PROCOL : TASKING

1) Définition

A task is a set composed by a program and eventually subprogramm, the run of which performs a certain function for the user.

A task has some characteristics defined at the generation of the system.

. A name
. A priority which can be of hardware level
or of software level

2) Restrictions

A task is not reentrant
A task, when running, must be **entirely** resident in one piece in core mémory
A task must be executed from the same address (T.2000 option)

3) States of a task

a) ACTIVE. .

the task is in the queue corresponding to its own priority and is given the control when all its predecessors have left the queue.

b) CANCELLED

the task is suppressed from the queue where it was waiting for its turn of having the control. This cancellation may be ordered from another task.

./.

c) TERMINATED

the task is normally terminated and control is given back to the moniter

d) WAITING

the task is interrupted (itself) and is waiting for an (or a sequence of) events

4) Task Scheduling

Among all the task present at the same time in core memory, the user has the possibility that not two tasks may be executed at the same time.

Two levels of scheduling:
'Hardware' tasks are scheduled at hardware level
'Software' tasks are scheduled at software level.

5) Task synchronization

the tasks use two types of ressources :

1 : physical ressources : processor, core memory....
2 : logical ressources : external sub-programs,

The executive PREX has to solve all kind of conflits.

Each ressource is assigned a semaphore ; tasks may ask for the ressource through the instruction :

./.

REQUEST SEM(i) : if the ressource is free, the task take it and the semaphore is set up to the busy state. If the ressource is busy, the task enters the queue of the semaphore SEM (i).

When the task has finished to use the ressource, it use the instruction

RELEASE SEM(i) Which frees the ressource and allows one of the task of the queue to access the ressource

Nota : the user may associate a semaphore to a virtual processor : when the semaphore is busy none of the task using this virtual processor could be activated.

-=-=-=-=-=-=-=-=-=-=-=-=-

LTPL EUROPEAN GROUP
LTPLE / 018

Category I, D.6
Update none
Obsolete none
2 PP

G. LOUIT
STERIA-FRANCE
20 December 71

PROCOL : INPUT / OUTPUT

A distinction has been made in PROCOL between classical and industrial input/output

1° Classical I/O

As in FORTRAN

2° Industrial I/O

The user has the possibility to refer symbolically by an unique identifier to an industrial peripheral or a part of it.

a) General format of the instruction

ORDER (⟨peripheral⟩, ⟨format⟩, ⟨complementary information⟩)list of variables.

b) Execution

- in input : read the information on the peripheral, following the indications of the format and transfert into the data zone named in the list.

- in output : transfert of the information from the variable to the peripheral according to specified formats.

./.

c) Description of i/o in the COMMON part

- i/o description

  type : analogic, digital, input, output, special

  number of module using the defined type

  specification of channels of the used module

  name assigned to the described i/o unit by the user

- format description

  different field

  check on now measures

  filtering

  linearization, conversion

  check on logical information.

There is the possibility to add new types of I/O (type SP).

LTPL EUROPEAN GROUP
LTPLE / 019

Category : I, D.4
Update none
Obsolete none
1 p

G. LOUIT
STERIA-FRANCE
21 December 71

PROCOL : TIMERS HANDLING

The user is allowed to handle a certain number of TIMERS.
A timer has a name and a step ; This is declared by '/TIMER' when generating the system.

Instructions

START : initializes the timer and sets up its value to zero
STOP : stops the timer.
AFTER : asks for the execution of a particular job after a certain number of step of the timer.
EVERY : asks periodically for the execution of a particular job.

-=-=-=-=-=-=-=-=-=-=-=-

G. LOUIT
STERIA-FRANCE
March 27th 1972

LTPL EUROPEAN GROUP
LTPLE / 032.

Category : T
Update : LTPLE/018
Replace : none
5 PP

WORKING PAPER

INDUSTRIAL INPUT - OUTPUT in PROCOL

Procol has facilities for peripheral description which allow the user to refer to partcfor the whole peripheral by only one identifier (which may be an array). For exemple : one or several channels in a analogic module may be refered by one identifier.

1/ Types of industrial input/output

There are 8 types of input-output :

ANIN : analogic input module
ANOUT : " output
DIGOUT : digital "
DIGIN : " input
TSIN : transfer station input
TSOUT : " " output
SEQUIN :
SEQUØUT :

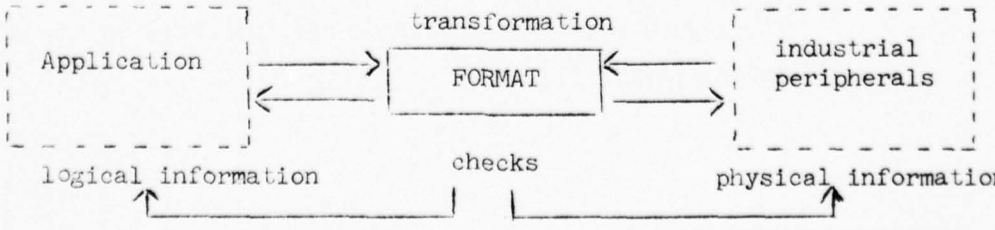
An industrial unit is a set channel-module to which the user gives a name.

2/ Description of industrial units

```
ANIN   (3) CHANNEL (5) TEMP 1, CHANNEL (A(12 : 25)) TEMP2 ;
ANIN   (5) CHANNEL (I : 8) PRESSURE ;
DIGOUT (7) CHANNEL (J)  DEBIT ;
```

The table A has been defined in the COMMON part.

3/ Description of the formats



./.

An industriel 1/O Procol will contain a certain quantity of information which allow the transformation of a logical information into a physical information and vice-versa. This transformation will be accompanied by various checks on the validity of the informations.
For example : an analogic acquisition needs three steps :

i) measure : - getting the raw mesure (physical information)
- check of the mesure validity
- filtering

ii) linearization-conversion : from the raw mesure to the corrected measure (user)

iii) check of the corrected measure

Example :

```
10 : IFORMAT  PHYSCHECK  ST (a, b, SP1 (REPORT, PARAMETER)),
              FILTER LT (15),
              CONV SP2 (LST, BUF, parameters),
              LOGCHECK BT (A, SP3, (LST,...) : B, SP4 (LST)) ;
```

< FORMAT > := < identification area > < control on physical information >
< filtering of physical information > < linearization-conversion >
< control on logical information >

i) Identification area
- adress of the format
- input or output

ii) Control on physical information
physical information : = information send to (from) the converter

in input : the raw measure
in output : the value to be displayed

The user has two possibilities for the check on the physical information :

./.

1 - Standard Option

the value is compared with the two bounds min and max

PHYSCHECK ST (min, max, SPE (REPORT, ---)) ;

2 - Non-standard option

the user specify his proper processing by means of a subroutine (SPE)

PHYSCHECK SPE (LST parameters, REPORT, ------)

In both cases the REPORT option reports on any misfonctionnement and the user can pursue or not the interpretation of the format.

Note - REPORT : = < Symbolic unit number >

< Identification of the channel which has created the overpass on a boud >

< Pick up or transmitter unavailability >

iii) Filtering of the physical information

The user has here two possibilities standard (a) and non-standard (b)

a) FILTER $\begin{Bmatrix} G \\ L \\ B \end{Bmatrix}$ (n)

n = rate

G = the raw measure must remain greater than $(1 - n\%).m$ (where m is the value of the most recent measure).

L = " " " less than $(1 + n\%).m$

B = " " " between $(1 - n\%).m$ and $(1 + n\%).m$

b) The non-standard option allows the user to specify his own mode of filtering by means of another subroutine.

FILTER SPE (LST parameters)

./.

iv) Linearization - conversion

This transformation is to be made by the user by means of a subroutine

CONV SPE (LST parameters)

v) Check of the logical information

The user has two possibilities :

a/ standard option :

LØGCHECK $\left\{ \begin{matrix} G \\ L \\ B \end{matrix} \right\}$ (bound$_2$, SPE$_2$, (...)) ;

'cm' means : corrected measure

G : bound$_1$ > bound$_2$

- cm > bound$_1$: nothing
- bound$_2$ < cm < bound$_1$: execute SPE$_1$
- cm < bound$_2$: execute SPE$_2$

L : bound$_1$ < bound$_2$

- cm < bound$_1$: nothing
- bound$_1$ < cm < bound$_2$: execute SPE$_1$
- cm < bound$_2$: execute SPE$_2$

B : bound$_1$ < bound$_2$

- cm < bound$_1$: execute SPE$_1$
- cm > bound$_2$: execute SPE$_2$
- bound$_1$ < cm < bound$_2$

b/ Non standard option :

The user may specify the treatement by means of a subroutine

LOGCHECK SPE (LST parameter) ;

4/ Industrial Input-Output Instructions

4.1 - General form

$\left\{ \begin{matrix} \text{INPUT} \\ \text{OUTPUT} \end{matrix} \right\}$ (<Industrial unit> [, < format number > < complementary information >]) List of variables

[ ] : means optional

4.2 - Analogic Input-Output

a/ single : INPUT (TEMP 1, n) ALP ;
acquisition from the industrial unit TEMP 1 under format number n and assignment to ALP

b/ grouping : INPUT (PRESSION, K) TABLE (10 : 17) ;
acquisition from the 8 channels of the industrial unit PRESSION under format number K and assignment of these 8 measures in TABLE (10), ---, TABLE (17)

c/ complementary information
there is different options :

i) GAIN of the converteur : RANGE < constant or integer variable >

ii) FILTERING : FILTER < precedent measure >
by default there is a standard treatment.

4.3 - Digital Input-Output

There is no format used

a/ single : industrial unit = 1 channel of a digital module
list of variables = 1 variable : item or table of bits

INPUT (CLE) ALPHA (J) ;

CLE is for example channel n° 3
ALPHA (J) is the j^{th} element in a table of bits

b/ grouping : industrial unit = m channels from a digital module
variable = word, structure, implicit loop of table of 1 bit item.

INPUT (CLES) BIT (5 : 8) ;

G. LOUIT
STERIA - FRANCE
March 27th 1972

LTPL EUROPEAN GROUP
LTPLE / 052
Category : T
Updates :
Replaces : nome
7 pp

WORKING PAPER

TASKING

IN PROCOL

In this section we refer to the "Tasking Facilities" part of the paper from LTPLA, numbered X3J1.4-27.

The states idle and dormant are not separated in PROCOL, because once a task has been created, a segment and a data area are assigned to it.

The other states are approximateley the same in Procol.

The most important new tasking feature appearing in Procol is the distinction between Softare Tasks and Hardware Tasks.

They are all declared in the same manner at the generation of the system.

TASKING

1. Definition

A task is a set composed by :

- a program
- and eventually subprograms

the run of which, performs a certain function for the user.

2. Task generation

A task is generated in the fallowing manner :

/TASK
< taskname 1 > : [priority-number] [, residence option] [, SEM (number)] where :
taskname is the identifier of the task .
priority-number :
residence-option may be : resident incore memory or net resident
(number)is a number assigned uniquely for each task and is used for the control semaphore .

3. Characteristics of a task

- Its name.
- Its priority which can be of hardware level or of software level
- An attribute for its residence.

a/ Conditions for execution of a task

The central unit is considered as a shared ressource for the tasks .
The user has the possibility to define which set of tasls will be sharing the C.P.U ;
he may decide that a task will not started if another task of the same set is running.

./.

b/ Conditions for interrupting a task

The user may specify the condition of preemption of a task by another with higher priority :

. left task : the task will be restarted only by an activation
. saved task : restarded from the point of interruption

4. Task operation

- ACTIVATE Taskname : the task is placed in the queue of the awaiting task for execution, according to its priority. If taskname has a higher priority, it preempts the task in execution.

the are two kinds of activation : conditionnal and inconditionnal.

- CANCEL Taskname : suppression of taskname from the queue

- EXIT : the task in execution is ended.

- WAIT : the task is interrupted and waits untill the wait appears.

ANNEX

FUNCTIONS FOR SOFTWARE TASKS

	FUNCTIONAL OPERATION	STATE CHANGE OF OBJECT TASK	ARGUMENT REQUIRED
GENERATION	CREATE	0 —>2	< TASK >
SYSTEM ACTIONS	CANCEL	2,3,4,6->2 or 5	< TASK >
	EXTERMINATE	can be done	< TASK >
	READY	5 —>3	< TASK >
	RUN	3 —>6	< TASK >
	PREEMPT	6 —>3	< TASK >
	INHIBIT	6 —>4	< TASK >
	UNSUSPEND	4 —>3	< TASK >
SELF ACTIONS	SCHEDULEME		< time>/<event >
	MYSTATUS	scheduled	
	EXIT	6 —>2 or 5	< TASK >
	DELAYME	6 —>4	< #TOPS> < TIM (n)>
	WAIT	6 —>4	< event><tops >
	SECURE	6 —>4 or 6	< semaphore >
ACTIONS ON OTHERS SOFT TASKS	CANCEL (EXECUTE)	2 —>3	
	SCHEDULE	2 —>5	< #tops> < TIM (n)>
	CANCEL (TERMINATE)	6,3,4->2 or 5	< task > < time>/<event>
	RELEASE	4 —>3	< semaphore >
	SIGNAL	4 —>3	< event >
	STATUS	scheduled	< task >

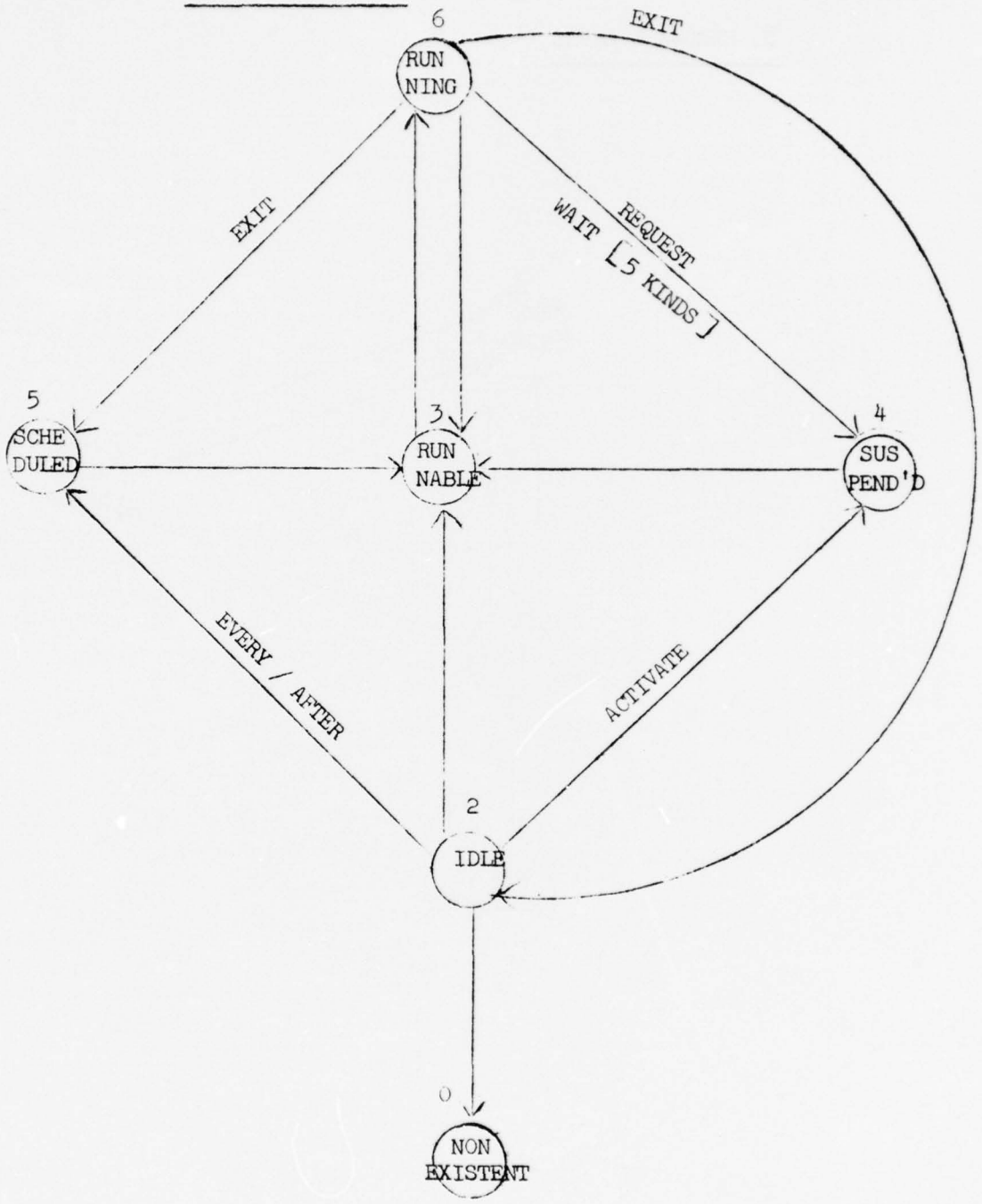
There are not the following functions :

exterminate
delete
Killme
Delay (another task)
Continue (another task)
Kill
suspend (another task)

The assigment of variable priority to other task or to itself is not allowed.

ANNEX

1. SOFTWARE LEVEL



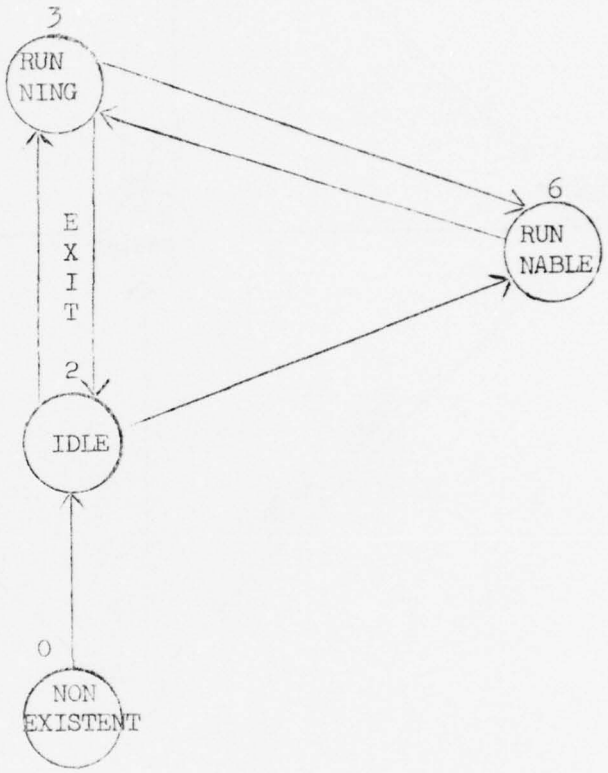
From any state exept 5, it is possible to come back to 2 by CANCEL

From any state it is possible to go to 5 by EVERY or AFTER

State vector : primary state / Priority / secondary state.

ANNEX

2. HARDWARE LEVEL



ANNEX

FUNCTIONS FOR HARDWARE TASKS

GENERATION	CREATE	0 —> 2
SYSTEM ACTIONS	RUN PREEMPT	3 —> 6 6 —> 3
SELF ACTIONS	MYSTATUS EXIT	6 —> 0
ACTIONS ON OTHER (SOFT) TASKS	EXECUTE SCHEDULE TERMINATE SIGNAL STATUS	2 —> 3

PRECEDING PAGE BLANK-NOT FILMED

G. LOUIT
STERIA-FRANCE
March 29th 1972

LTPL EUROPEAN GROUP
LTPLE/053

Category T
Updates : none
Replaces : none
5 pp

WORKING PAPER

HARDWARE - SOFTWARE LINKAGE

IN PROCOL

WORKING PAPER
HARDWARE-SOFTWARE LINKAGE
IN PROCOL

I) INTRODUCTION

This working paper describes the PROCOL solution for hardware-software linkage (T 2000 implementation) In PROCOL, the user generates the system means of a conversation with the Program Generator : PROG.

A "programmed application" is composed of two parts :
A set of "PROCOL instructions" corresponding to the different tasks and to be translated by the compiler PROC.
A set of "directives" describing the hardware configuration and the software organization to PROG.

2) GENERAL DESCRIPTION

< programmed application> := <P.i. set> < D. set>
<P.i. set> is the set of Procol Instructions
<D. set> := < COMPUTER CONFIGURATION > < APPLICATION DESCRIPTION>

2.1. < COMPUTIER CONFIGURATION > := PHYSICAL CONFIGURATION < P.C. QUERY LIST>
SYMBOLIC CONFIGURATION < S.C. QUERY LIST>
UTILISATION CHARACTERISTICS < U.C. QUERY LIST>

The mode of description is conversationnal and is expressed by a list of queries issued by the system generator to which the user in the site answers.

a) Physical configuration query list

List of priority levels for i/o
list of interrupts levels at which tasks with i/o are executing
Bac canal i/o list

./.

a.1.) Description of classical i/o

- system teletype
- other classical peripherals list
- buffer size (for i/o with format)

a.2.) Description of industrial péripherals

-Digital Input
- Digital Output (grouped or not)
- Analog Input
  - . Standard (query for each kind)
  - . non standard
- Analog Output (query for each kind)
  - . Standard
  - . Non standard

b) Symbolical Configuration Query list

Classical Symbolic Unit list
Industrial Symbolic Unit list
Digin module
digout module
sequin module
sequout module
anin module
anout module
tsin module
tsout module

c) Utilisation characteristics

- Industrial formats : Residence Place, size, working area number
- Conversion Problems for Industrial Formats
- Different other uses (Semaphores, wait, Every, After, Mask, Signal

./.

2.2. < APPLICATION DESCRIPTION > := < Tasks Information query list >
< Interrupts Information query list >

a) Task Information Query list includes questions such as :

Max number of tasks
Names of the resident tasks
Names of the non resident tasks
Names of external reentrant subroutines
Description of tasks
Description of timers, pulses
Semaphore handling
Memory dynamic management

b) Interrupts information query list includes questions such as :

- hardware level
- event level
- software level

} $It_j \longleftrightarrow$ TASK T_j

2.3. Facilities query list

There are other question for modifying, updating and loading, independently of any application

3) INDUSTRIAL I/O PERIPHERALS AND FORMATS DESCRIPTION

i/o peripherals and format are described in a the COMMON part of the application program by means of special declaration instructions.

BEST AVAILABLE COPY

./.

a) i/o peripheral description

This has been detailed in LTPLE/Ø32A : "INDUSTRIAL INPUT/OUTPUT in PROCOL" In Procol several types of i/o have been distinguished. For each type a keyword has been assigned, e.g. "ANIN" means Analogic Input Module.

The user has the possibility to address by a symbolic name all or part of a peripheral devices ; this name designate an industrial unit.

Each industrial unit has to be described by special instruction which include type, device, channel, and name of the i. unit. A special possibility of PROCOL is to designate by a single name a set of channels of a module which are not necessary consecutive.

b) i/o format description

Described in details in LTPLE/032 A

4) ADVANTAGES OF PROCOL

- The separation in two parts yields to a better efficiency in the use of the machine and in the use of the language .
- The conversation mode for describing the configuration is very easy to use with, in some cases, pre-formated answers.
- In case of changing computer and its environment for the same application, the reprogramming effort is only limited to the hardware dependent part.
- The possibility to handle a specified set of channels in a module provide much more conveniency.

BEST AVAILABLE COPY

PRECEDING PAGE BLANK-NOT FILMED

U.D.C. 800.92:681.323.06: :62-52

Development of Process Control Language—PCL

Hiroshi Kuwahara*
Toshihiro Hayashi*
Kazuhiko Fukuoka**
Tadaaki Bandou***

ABSTRACT: A higher procedural language for use with the Hitachi control computer 500 has been developed at Hitachi and named Process Control Language – PCL. The PCL has been developed based on a study of functions required for process control, and it adopts not only a number of essential functions such as structural description of data, bit handling, decision table processing, virtual-memory-like data processing, communication with operating system, but also many powerful debugging facilities and powerful optimizing methods to make object codes shorter. Consequently, as compared with the conventional programming using assembler languages, this language has brought about a remarkable improvement in productivity, visibility, uniformity and maintenability of software with little sacrifice on the part of memory capacity and running speed of the object codes.

INTRODUCTION

WITH the full-scale application of computes to plant automation and labor-saving in industry, the amount of programs required for each application is becoming enormous. In the field of computer control, there is even talk of a "software crisis."

A control program should meet the following requirements:

(1) To be compact, because many computers are of relatively small scale and the main memory is subject to great restrictions.
(2) To have a high execution efficiency, as most programs are fixed after completion of a project.
(3) Since many programs are for logic decision processing, easy and accurate coding for such processing should be ensured.
(4) It should permit easy and accurate coding for on-line real-time problems utilizing multiprogramming capabilities.

These requirements cannot be satisfied by conventional compiler coding, so that assembler language has been used chiefly. In order to overcome the "software crisis", one of the best way is to replace the assembler language with a higher level language that has the necessary and sufficient functions, thereby improving the productivity, visibility, uniformity and maintenability of software. Process control language (PCL) we developed is a high-level language meeting these requirements. Its functions and features will be described.

*Omika Works, Hitachi, Ltd.
**Central Research Laboratory, Hitachi, Ltd.
***Hitachi Research Laboratory, Hitachi, Ltd.

POLICY OF DEVELOPMENT

Position of PCL

PCL holds a place at the top of the language processing system of the Hitachi control computer 500 shown in Fig. 1. In the Hitachi control computer 500 language processing system, the programming languages are PCL, FORTRAN, and assembler. FORTRAN is used for making up special calculation programs containing complex number, such programs being very rare in the control field. The assembler language is used for making up reentrant subroutines resident in the main memory and used in common by a plurality of tasks. Other than these special programs, most of the control programs can be made up by PCL. The usage rates of the three programming languages are shown in Fig. 2.

Language specifications

In developing PCL, the language specifications were determined according to the following guidelines:

(1) It should have sufficient language functions for control purposes, without combined use of assembler language. – Some so-called control compiler languages permit mixed use of assembler language, but this is hard on operators who are accustomed to compiler language. On the other hand, this tends to make those accustomed to assembler language slight compiler language. At any rate, the features of a higher-level language cannot be fully taken advantage of.

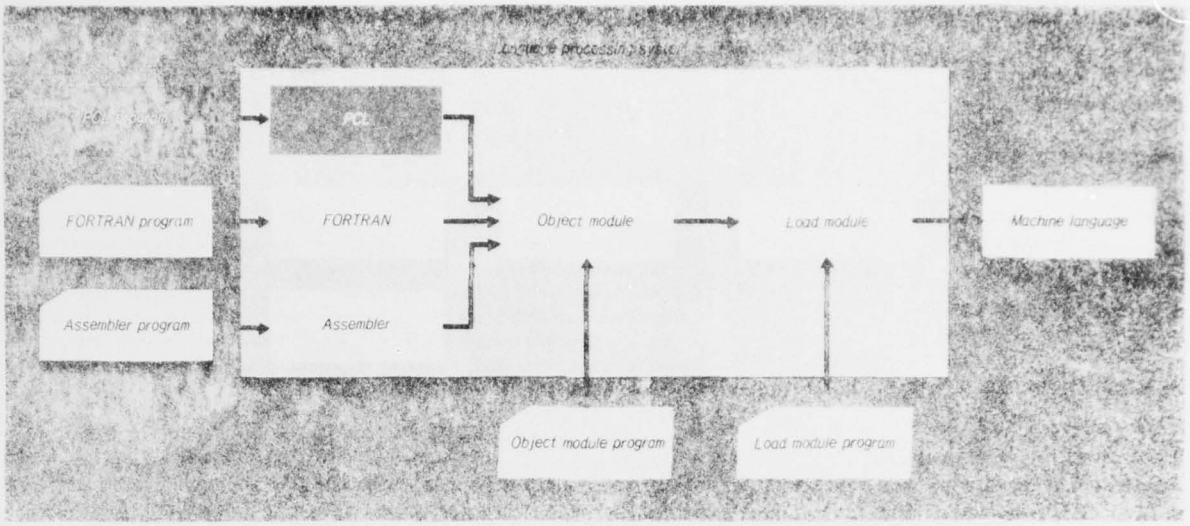


Fig. 1—Hitachi control computer 500 language processing system. PCL, FORTRAN, and assembler are languages for program coding. The binder combines the programs in these different languages into a larger load module program. The loader changes the load module program into a form (machine language) that can be executed by the computer.

(2) The language system should be such that previously developed programs coded in assembler language can be effectively utilized. – Despite (1), it is important that formerly made assembler coding programs be effectively used. Generally, linkage of a subroutine made up of assembler language involves a large volume of work when the difference from compiler language is to be overcome by modifying the already made subroutine.

(3) A familiar language system. – Since FORTRAN is most widely used in Japan, the new language system should conform as far as possible to FORTRAN statements.

(4) An easy-to-use language system. – In addition to achieving the required functions, the language system should ensure ease of coding and use.

(5) A language system that effectively utilizes the multiprogramming functions of an operating system (OS). – Even if the individual programs are highly efficient, it would be meaningless if the entire system had a poor efficiency. Therefore, the language system should permit effective utilization of OS functions, above all, multiprogramming functions.

(6) A language system that takes debugging into consideration. – It is said that debugging accounts for more than half of the work for developing a program. Thus, the language system should readily lend itself to debugging.

(7) A language system that has reinforced input/output functions. – A computer control system uses numerous special input/output devices, such as process I/O

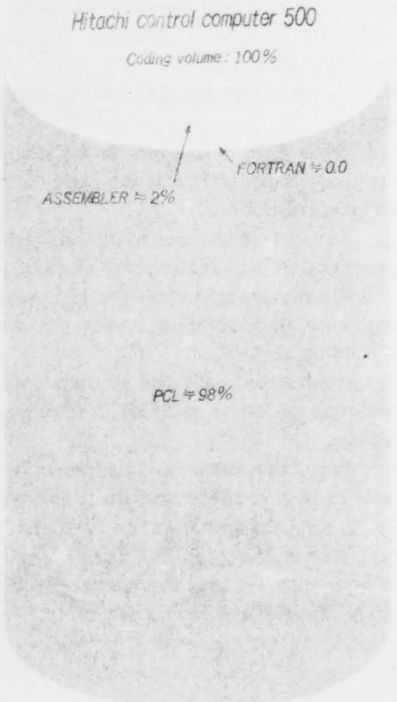


Fig. 2—Usage rate of programming languages. About 98% of a control program is made up in PCL. Of the remaining 2%, special programs including complex number are in FORTRAN, and reentrant subroutines resident in the main memory and used in common by a plurality of tasks are in assembler language.

devices and color cathode ray tubes. Therefore unless efficient coding can be performed for these, the language system cannot be called satisfactory for process control.

Compiler performance

Factors for evaluating compiler performance are: (1) time required for compiling, (2) size of object, and (3) executing time for object. In PCL, (2) and (3) are given priority over (1). Priority of (2) and (3) is determined differently in each statement, after careful justification study.

FEATURES OF PCL

We analyzed the characteristics of programming of process control programs, then set the target values of functions and performances. These targets were attained in PCL. Therefore, while individual statements could take optional forms, considering familiarity according to the guidelines, we tried to make the coding format as close as possible to that for FORTRAN. However, the syntax, in order to realize the original target, represented a considerable expansion from FORTRAN. Major features of PCL will be described in the following.

Features in specifications

Specification statement permitting easy use of integer type data: In control programs, as is seen in accumulated data processing, accumulation error and rounding error often develop if integer operation is not performed. In the case of FORTRAN, data of integer type by implicit type specification are only those alphanumerics starting with I, J, K, L, M, and N. For data starting with other letters, statement of integer type must be made individually, causing much trouble. In the case of PCL, by setting an IMPLICIT statement, the rule of implicit type specification can be expanded at will, so that integer type specification can be made without individual statement,

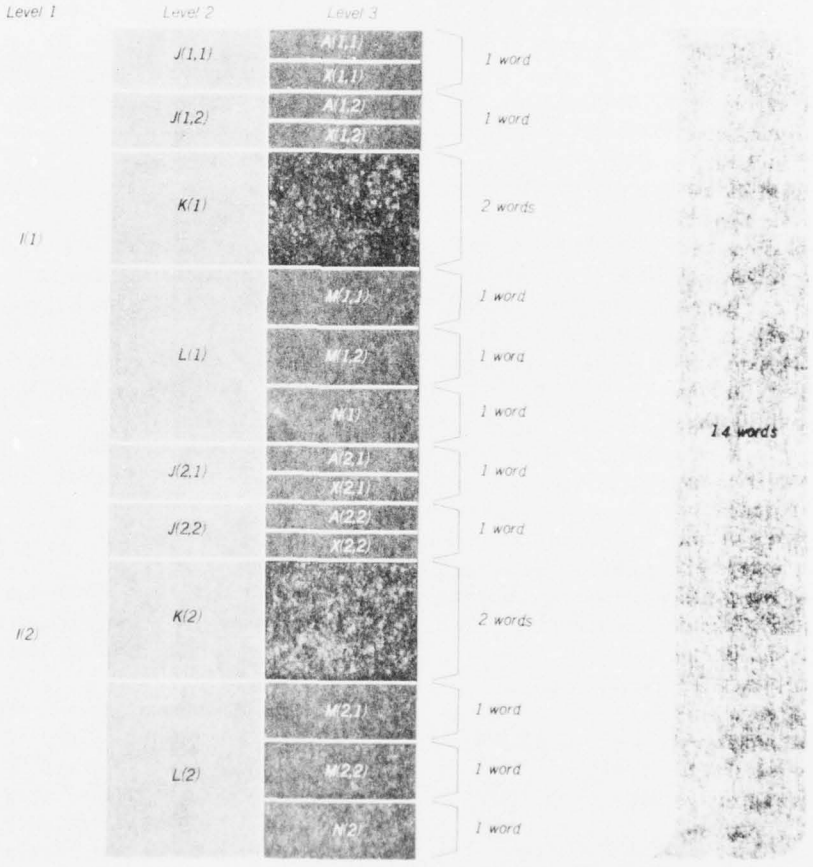


Fig. 3–Memory layout of data structure.
I consists of the two sets I(1) and I(2) (level 1). Each I further consists of two sets of J and K, and one set of L (level 2). J consists of 8-bit data in which each bit is A, X, while L consists of two sets of M and one set of N (level 3). In total, a 14-word memory is formed.

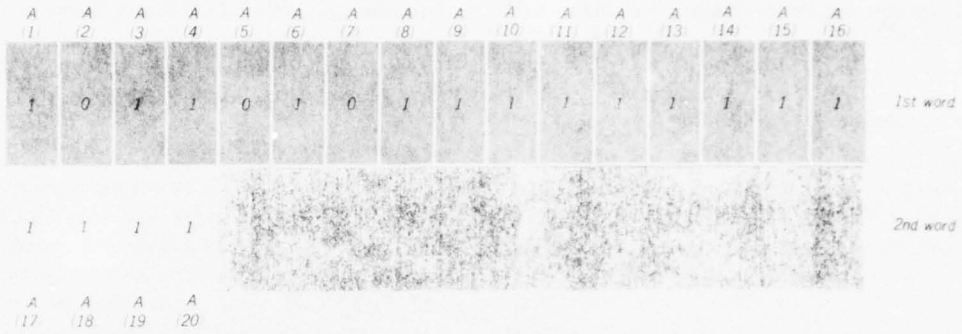


Fig. 4—Bit dimension and bit data.
A has an area of 20 bits, from A(1) to A(20). Initial value for A(1) is 1, A(2) is 0, ... A(20) is 1. For the part corresponding to A(21) through A(32), areas are secured but not defined, so that they cannot be used.

even for alphanumerics starting with letters other than I, J, K, L, M, and N.

⟨Example⟩
```
IMPLICIT INTEGER (A – N)
IMPLICIT INTEGER* 2(O – X)
```

By the above statement, all variable names and array names starting with any letter in A, B, C, ... N will be regarded as of single precision integer type, and all variable names and array names starting with any letter in O, P, Q, ... X will be of the double precision integer type, so that only those variable names and array names starting with Y or Z will be of the real type.

Conversely, if the statement is made as follows:

```
IMPLICIT REAL (A – K)
```

variable names and array names starting with A, B, C, ... K and O, P, Q, ... Z will be of the real type, and the only variable names and array names of the integer type are those starting with L, M, or N.

Coding for hierarchy data by structure specification: Various tables usually consist of a number of data elements, each of which is made up of one or two words or several bits. Each of the data elements and each of the sets of data elements are named, so that reference can be made via data element name and data set name. In this way coding for hierarchy data by structure specification is made. This is known as data structure as in PL/I.

⟨Example⟩
```
1I (2),
    2J (2),
        3A:BIT (8),
        3X:BIT (8),
    2K (2),
    2L,
        3M (2),
        3N,
```

The memory layout of the above data structure is shown in Fig. 3.

Abundant bit processing functions: For processing of 1-bit data (represented by contacts and switches), processing of decision tables (represented by sequence control), and processing of data of several bits' length (often seen in internal processing of program), coding can be made systematically at the same level as ordinary data.

⟨Example⟩
```
BIT DIMENSION A (20)
BIT DATA A/10110101, 12*1/
```

In this example, by BIT DIMENSION statement, array element names from A (1) to A (20) are given in units of bits; and for each, by BIT DATA statement, initial values are given so that A(1) will be 1, A(2) will be 0, A(3). . .A(8) will be 1, and A(9). . .A(20) will be 1. This is illustrated in Fig. 4.

For such statement of BIT type array, there are in addition BIT COMMON statement and BIT GLOBAL statement. All statements can specify up to 3rd dimension.

⟨Example⟩ Decision table processing

The 20-input 5-step decision table shown in Fig. 5 is coded as in Fig. 6.

⟨Example⟩ Bit string expression

When A, X, and N are those in the data structure of Fig. 3,

```
N(1) = A(1, 1) & X (1, 1) + N (2)
```

indicates that on the assumption that the result of taking AND in bits for A(1, 1) and X(1, 1) is a positive integer, the result of adding N(2) to this positive integer is substituted into N(1).

Bit string expression is generally defined as follows:

```
When  I :   1100110011001100
        J :         11110000
then ¬ I :  0011001100110011
     I & J :0000000011000000
     I | J :1100110011111100
     I % J :1100110000111100
```

Statement for data straddling tasks: In a control computer, usually a plurality of programs are operated by multitasking, and an area for exchange of data between tasks is needed.

BEST AVAILABLE COPY

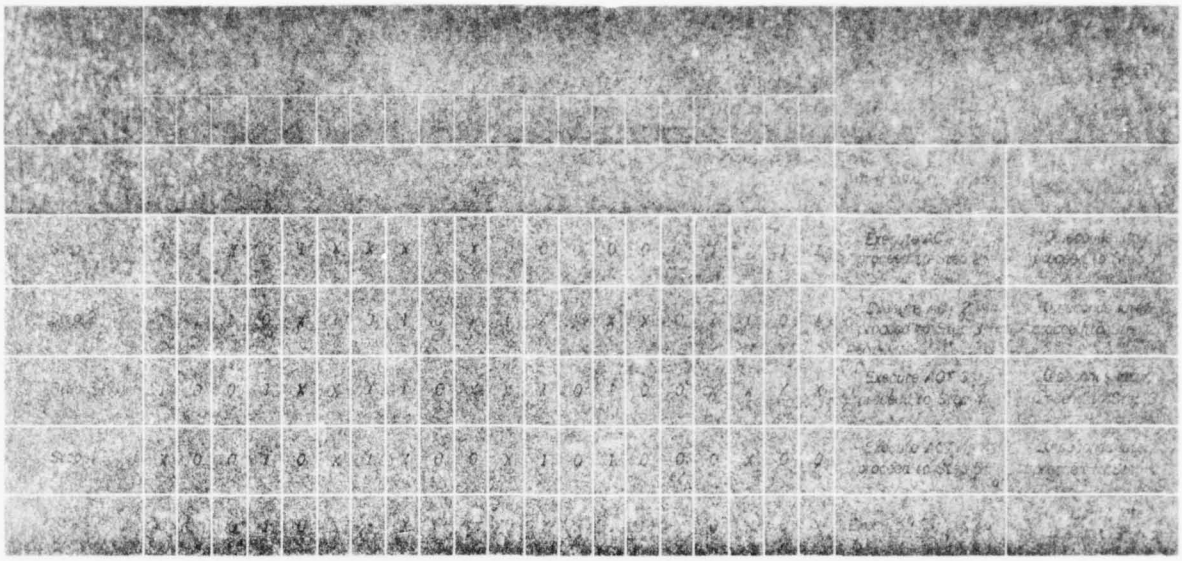


Fig. 5–Example of decision table.
In Inputs 1 through 20, various information converted to two-value information of 0, 1 by separate tasks is accommodated at specified time intervals. In "Condition" column, 1 means corresponding input must be 1, 0 means it must be 0, X means it may be 1 or 0. At that time, the decision table checks the whole of input A and each step of Condition C, bit-by-bit, and depending on whether the condition holds, processing shown in the "Action" column is performed.

```
PAGE 0101 HIDIC 500  COMPILE  LIST
 EFN                          PCL SOURCE STATEMENT
 G  DECISION TABLE PROCESSING TASK  DECTK
      IMPLICIT INTEGER    A-Z
      BIT DIMENSION       C(5,20)
      DECISION DATA       C(1)='11X01XXXXX0000011111',
      1                   C(2)='11110X10101XXXXX01101',
      2                   C(3)='100110XX1100X10100XXXX',
      3                   C(4)='1X0010X1X00X101000X00',
      4                   C(5)='100010XXX000XX10X0X0X'
      BIT GLOBAL  A(20)
      VALUE    TI,DECTK
C  N1 = TIMER NO.,  TIMER UNIT = 0.1SEC, YUENI   10SEC = 1.00
C
      DO 50 I= 1,5
   10 DECISION IF (C(1,EQ.A) GO TO (1,2,3,4,5),I
      TIMER    T1, 100, DECTK
      WAIT
      GO TO 10
    1 CALL ACT1,&50
    2 CALL ACT2,&50
    3 CALL ACT3,&50
    4 CALL ACT4,&50
    5 CALL ACT5,&60
   50 CONTINUE
   60 STOP
C  CALL ACT,&50 WA SUBROUTINE NO JIKKO NO ATO
C                        50 E JUMP SURUKOTO O SIMES
      END
```

Fig. 6–Example of coding for decision table processing.
This shows a coding of the decision table in Fig. 5 by using PCL statements. Input A shows that it is held in core resident area by BIT GLOBAL A(20). Condition Steps 1 through 5 show that areas and initial data are taken in this task by BIT DIMENSION C(5, 20) and DECISION DATA C(1) through C(5). These inputs and conditions are checked by DECISION IF statement indicated by statement number (EFN) 10.

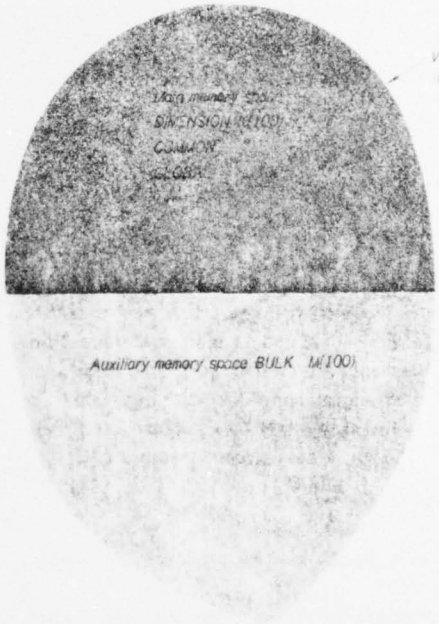


Fig. 7–Concept of virtual memory space.
Data can be processed without distinguishing main memory and auxiliary memory, but by regarding the whole as a single memory (virtual memory) space.

BEST AVAILABLE COPY

HIDIC PCL CODING SHEET

PROGRAM | CODED BY | DATE : | DECK ID | PAGE | OF

COMM	STATEMENT NUMBER	CONT	PCL STATEMENT	DECK ID	PAGE	SEQ NO
			C VIRTUAL MEMORY SPACE			000
			DIMENSION K(100)			050
			GLOBAL L(100)			100
			BULK , M(100)			150
			DO 100 I=1,100			200
	100		K(I)=L(I)+M(I)			250
			STOP			300
			END			350
						400
						450
						500
						550
						600
						650
						700
						750
						800
						850
						900
						950

Hitachi, Ltd.

Fig. 8 Example of coding using virtual memory data space. By adding data L(1) in core resident area and data M(1) in auxiliary memory, the program to be accommodated in area K(1) in the task can be written as K(1)=L(1)+M(1), so that there is no need to think of the data location in the expression.

This area is usually secured in the resident area on the main memory or on the auxiliary memory. Heretofore, reference to this area has been made by absolute addressing (assembler), or reference has been impossible (FORTRAN). In PCL, it suffices to state the required area as a variable name or an array name; assignment to the main memory or the auxiliary memory is done automatically by a loader (Fig. 1).

〈Example〉 GLOBAL Statement
BIT GLOBAL statement

For these, data areas are secured in the residence section of the main memory.

BULK statement

For this, data areas are secured on the auxiliary memory.

Data with a concept of virtual memory: Even when a systems engineer or a plant engineer, to say nothing of a specialist in programming, is making up a program, he will not have to be especially conscious of data on the auxiliary memory (drum memory) and can handle them directly within various statements in the same manner as data on the main memory (core memory). Fig. 7 illustrates the concept of virtual memory space adopted in PCL, and Fig. 8 an example of coding using virtual memory data space.

Incorporation of system communication statement: Generally, in an on-line real-time control system, in order to effect synchronization of the various tasks being operated independently, it is necessary to stop, operate, or temporarily stop certain tasks, and to prevent illegal sequence of data read/write or deadlock for resources used in common by various tasks; exchange of signals for this is all conducted via OS control instructions.

In PCL, the function of OS control instruction is incorporated as system communication statements.

〈Example〉 QUEUE statement, START statement,
STOP statement,
WAIT statement, RESERVE statement,
FREE statement, ...

Abundant and expandable I/O functions: For color CRT and process I/O devices (e.g. AI, DI, AO, DO), as well as general I/O devices (e.g. L/P, C/R, PTR, PTP), coding can be made in the same statement and in the same concept. Moreover, consideration is given so that the peculiarities of individual I/O devices do not appear in the language.

Fig. 9 is an example of coding for color CRT display. If the "70" in the WRITE statement in Fig. 9 is the file reference number for color CRT display, then, by

```
C  COLOUR CRT DISPLAY
       INTEGER  K
       K 1000
       WRITE (70,100) K
 100 FORMAT(!RED,I5)
       STOP
       END
```

Fig. 9–Example of coding for color CRT display.
If "70" in WRITE statement indicates color CRT display, then by execution of the program, the value of K(1000) is indicated in red in integer type 5 digits.

```
C  DEBUG FACILITY (COMPILE MODE)
       COMPILE MODE  *
       GLOBAL  L(100),M(100),N
*      DEBUG VALUES  1,100
*      DEBUG SUBSCRIPT L,M
*    1 CONTINUE
       DO 100 I=1,N
   100 L(I)=M(I)+N
%      WRITE(6,200) L
% 200 FORMAT(1H ,2015)
       STOP
       END
```

Fig. 10–Example of usage of compile mode.*
Since compile mode is by specification, statements in which the first column is % are treated as notes.*

executing this program, the value of K(1000) is displayed in red in an integer type five-digit numeral.

Ease of linkage with existing programs of assembler coding: Comparison of the argument processing systems of a subroutine made up of assembler language and one made up of compiler language shows that in the former usually the argument value itself is set but in the latter the argument value itself is not set but rather the address where the argument value is accommodated is set. Therefore, successful linkage cannot be effected by calling an assembler-language subroutine as it is from a compiler-language program, so that usually the assembler-coding subroutine has to be changed.

With PCL, processing is possible for any linkage system, so that no revision is required even when an assembler-language subroutine is to be called from a compiler-language program.

〈Example〉 CALL SUB (A, 10)
CALL SUB (@A, @10)

When there is the @ mark in front of the argument, the usual compiler argument processing is not made, but rather the argument processing method usually employed in assembler language is used.

Abundant debugging functions: In addition to dump and trace functions, PCL has the compile made specification function to make switching between statements for off-line processing and on-line processing, the function to switch the subroutine used, and the overflow check function for array subscripts. Several hundred error messages are provided for use at compiling and execution.

〈Example〉 DEBUG VALUES statement

When specified values are given to variables and arrays within the range of the specified statement numbers, the variable names and array element names as well as the values substituted into them are printed out by L/P.

DEBUG FLOW statement

When a GOTO statement, arithmetic IF statement, or RETURN statement has been executed within the specified range of statement numbers, the shift of control is printed out.

DEBUG SUBSCRIPT statement

When in a specified array, the subscript of an array element name is larger than the stated value, printing is made to that effect.

COMPILE MODE statement

An example of usage of complie mode statements is given in Fig. 10. In this statement, WRITE statements and FORMAT statements are processed as footnotes. If in Fig. 10 the COMPILE MODE is specified by %, then the result of compilation will be as presented in Fig. 11.

RENAME statement

An example of usage of this line expression will be given below:

```
RENAME (FETCH, DUMP)
        ⋮
CALL FETCH (A, B, C)
```

In this example, CALL FETCH (A, B, C) will be executed as CALL DUMP (A, B, C).

Features in performances

Efficient object taking advantage of 16-bit machine: A 16-bit machine usually has 1-word instructions (16-bit instructions) and 2-word instructions (32-bit instructions). In PCL, 1-word instructions are used as far as possible, as a result of which efficient objects are formed. It is in this respect that conventional compiler programs

```
PAGE 0001 HIDIC 500 COMPILE  LIST
 EFN                          PCL SOURCE STATEMENT
C   DEBUG FACILITY (COMPILE MODE)
        COMPILE MODE %
        GLOBAL I /100/, M/100/, N
        DO  100 I = 1,N
      100 L,I = M(I) + N
%       WRITE(5,200) L
%     200 FORMAT(1H,2015)
        STOP
        END
```

Fig. 11–Example of compilation by compile mode %
*The program in Fig. 10 is compiled by specifying the compiler mode as %. All statements with * in the first column are neglected.*

for 16-bit machines are said to provide less efficient objects than programs in assembler language. In PCL, this problem is resolved, and efficiency is vastly improved.

Object aimed at efficient utilization of auxiliary memory:

Areas defined by variable names and array names, except those where the initial values are set by DATA stetements, are all work areas where the values are accommodated at execution, so that it suffices to secure the areas at program roll-in from auxiliary memory to main memory for execution. In conventional compiler language, thought is given only to moving the program in the main memory, with the result that, when an object formed is stored on auxiliary memory as an object for multiprogramming, work areas corresponding to the variables and arrays are secured also on the auxiliary memory. This is one cause of increasing the capacity of auxiliary memory. This problem is solved in PCL.

Common subordinate program: In conventional compiler objects, subordinate programs (e.g. READ/WRITE processing program at execution, type change program, various function modules) follow each main program. Generally, for two or more main programs (tasks in PCL) that are used in common, by making them reside in the main memory, the size of individual programs on the main memory and auxiliary memory can be vastly reduced. In PCL, therefore, all the subordinate programs made automatically in compiler language are provided in reentrant form, so that they can easily be made resident in the main memory and can be used simultaneously in two or more tasks.

Subordinate program of order reentry system: A number of subordinate programs are provided for a single function. For instance, there are two kinds of programs (same link name) which have identical functions, but one has a larger number of words but shorter execution time, and the other has a smaller number of words but a longer execution time. A user that employs the function at a high frequency may adopt the former program, while a user that employs it at a low frequency may adopt the latter program.

Making I/O processing programs into tasks: I/O processing programs in conventional compiler language constitute subordinate subroutines for I/O format and parallel element processing. This is devoid of the concept of multiprogramming in which parallel processing is made by CPU and I/O devices.

With PCL, this point is solved by making I/O processing programs an independent task. Also, the programs are made available for common use by a plurality of tasks without making them resident in the memory. Further, by buffering the exchange of I/O data, temporary high-traffic processing demand on low-speed I/O devices is smoothed.

Other optimization techniques: In addition to abovementioned techniques, various techniques for optimization at the statement level are adopted in order to minimize execution time and memory space. These techniques include optimization of DO LOOP, optimization of array subscript calculation, optimization of arithmetic IF statements, and optimization of GOTO statements.

By adopting the various optimization techniques described above it has been confirmed that: for main routines of individual programs, they are equal to or 10% larger than an assembler coding program made by an extremely experienced programmer and for a whole computer system including subordinate programs and OS, the memory size is about 10% larger than in cases where assembler language is used. Furthermore, if a program is made by a normally experienced programmer, the program size by PCL is often equal or even less in some cases.

CONCLUSIONS

With regard to the process control compiler language PCL developed for the Hitachi control computer 500, the guidelines for development, and features in language specifications and performances have been described.

Heretofore, programs for process control or on-line processing have automatically meant assembler coding programs. The authors see the PCL will open the way for extensive use of higher level languages in process control and on-line real-time processing.

At present PCL is designed for use with the Hitachi control computer 500, but work is under way to make it applicable to computer systems of the Hitachi control computer series. They are Hitachi control computer 150, 350 and 700.

The authors express their thanks to those who have cooperated in the development work.

