Aston University

Some pages of this thesis may have been removed for copyright restrictions.

If you have discovered material in Aston Research Explorer which is unlawful e.g. breaches copyright, (either yours or that of a third party) or any other law, including but not limited to those relating to patent, trademark, confidentiality, data protection, obscenity, defamation, libel, then please read our Takedown policy and contact the service immediately (openaccess@aston.ac.uk)

DIGITAL CONTROL SYSTEM PROGRAMMING
FROM PROCESS CONTROL DIAGRAMS

Thesis submitted to the

Department of Electrical and Electronic Engineering
University of Aston in Birmingham

for the degree of

DOCTOR OF PHILOSOPHY

by

CH'NG YI LINN
B.Sc., M.Sc.

JULY 1980

To my Father, Mother and Huey Yi

'Digital Control System Programming From Process Control Diagrams'

Ph.D. Thesis, CH'NG YI LINN, July 1980.

SUMMARY

A graphical process control language has been developed as a means of defining process control software. The user configures a block diagram describing the required control system, from a menu of functional blocks, using a graphics software system with graphics terminal. Additions may be made to the menu of functional blocks, to extend the system capability, and a group of blocks may be defined as a composite block. This latter feature provides for segmentation of the overall system diagram and the repeated use of the same group of blocks within the system.

The completed diagram is analyzed by a graphics compiler which generates the programs and data structure to realise the run-time software. The run-time software has been designed as a data-driven system which allows for modifications at the run-time level in both parameters and system configuration. Data structures have been specified to ensure efficient execution and minimal storage requirements in the final control software.

Machine independence has been accomodated as far as possible using CORAL 66 as the high level language throughout the entire system; the final run-time code being generated by a CORAL 66 compiler appropriate to the target processor.

KEYWORDS: process control, graphics, language, data structure.

ACKNOWLEDGEMENTS

I wish to thank all those who have helped in one way or another in this project, particularly the following -- Professor H. A. Barker for his initial supervision; Dr. G. K. Steel for his subsequent supervision and invaluable guidance; Dr. R. C. Johnson, Dr. R. G. Wilson and Mr. P. W. Davy for their advice and general concern throughout the period of our acquaintance; my co-researcher Mr. J. W. Lim for his co-operation; and not least, my fiancee Huey Yi and our families for their patience and understanding throughout the period of the project.

CONTENTS

Summary i

Acknowledgements ii

Contents iii

List of Illustrations vii

List of Tables x

Chapter 1 Process Control

1.1 Introduction 1

1.2 Programming Languages 4

1.3 Research Objectives 7

Chapter 2 Software Requirements

2.1 Introduction 10

2.1.1 Ease of Use 10

2.1.2 Flexibility 11

2.1.3 Promotion of Structured Programming Practice 12

2.1.4 Facility of Documentation and Maintenance 13

2.1.5 Portability 13

2.1.6 Efficiency 14

2.1.7 System Interrogation and Modification 15

2.1.8 Security 15

2.2 Definition of the Graphical Process Control System 16

2.2.1 Ease of Use 16

2.2.2 Flexibility 17

2.2.3 Promotion of Structured Programming Practice 18

2.2.4 Facility of Documentation and Maintenance 18

2.2.5 Portability 18

2.2.6 Efficiency 19

2.2.7 System Interrogation and Modification 20

2.2.8 Security 20

2.3 Software Generation Process 20

Chapter 3 Graphics

3.1 Introduction 22

3.1.1 Graphics Software 22

3.1.2 Graphics Hardware 23

3.2 Graphical Process Control Language (GPCL) 25

3.2.1 Macro Facility 27

3.2.2 Components of GPCL 27

3.2.3 Main Graphics Requirements 29

3.3 Graphic Data Structure 31

3.3.1 Design Criteria 31

3.3.2 Attributes of Simple Blocks 32

3.3.3 Single-level Data Structure 34

3.3.4 Multi-level Structure 39

3.3.5 Limitations of Macro Blocks 45

3.4 Graphic Editor 48

3.4.1 Hardware Description 48

3.4.2 Menus 50

3.4.3 Editing Aids 52

3.4.4 Editing of Blocks 53

3.4.5 Editing of Connections 55

3.4.6 Editing Text 57

3.4.7 Editing Simple Block Types 60

3.4.8 Editing Composite Blocks 62

Chapter 4 Functional Blocks

4.1 Introduction 64

4.2 Functional Characteristics 65

4.3 Procedures 68

4.4 Internal Representation of Variables 70

4.4.1 Storage Requirements 70

4.4.2 Speed 72

4.4.3 Range 72

4.4.4 Logical Variables 73

4.5 Some Implementation Considerations 74

4.5.1 Constant Pool Block 75

4.5.2 X-Y Function Block 75

4.5.3 Delay Block 77

4.5.4 Input/Output Interface Blocks 79

4.5.5 Analog Input/Output Blocks 79

4.5.6 Digital Input/Output Blocks 82

Chapter 5 Compilation and Run-time Structure

5.1 Introduction 83

5.2 Components of Control Program 84

5.3 Compilation From Graphics 85

5.3.1 Sequencing 86

5.3.2 Treatment of Macros and Subpictures 91

5.3.3 Generation of Control Program 93

5.4 Run-time Data Structure 95

5.4.1 Run-time Values Table 96

5.4.2 Block Table 98

5.4.3 TYPE Table 102

5.4.4 Accessing of Variables 106

5.4.5 Junction Blocks 110

5.4.6 Text Table 111

5.4.7 Machine Independent Representation of RDS 113

Chapter 6 Run-time System

6.1 Introduction 115

6.2 Run-time Data Structure Initialization 116

6.3 Function Block Processing 116

6.4 Sequencing 117

6.5 Operator Interaction 120

6.6 Additional Supervisor Functions 122

6.7 Interrupt Processing 123

Chapter 7 System Performance

7.1 Graphics 125

7.2 Run-time System 127

Chapter 8 Conclusions 129

Appendix 133

References 145

List of Illustrations

2.1	Software Generation Process	21
3.1	Dye-Mixing Process Block Diagram	26
3.2	Cascade Controller - Example of Composite Block	27
3.3	Block Diagram With Composite Block	28
3.4	Block Schematics	35
3.5	Components of Simple Graphic Data Structure	35
3.6	Simple Graphic Data Structure	36
3.7	Example of Composite Blocks	40
3.8	Multi-level Nature of Diagram	40
3.9	Components of Graphic Data Structure	43
3.10	Graphic Data Structure	44
3.11	Cascade Controller Block	46
3.12	Menu Substructure	51
3.13	Text Nodes	59
3.14	Temporary Line Drawing Data Structure	61
4.1	Example of Functional Blocks	64
4.2	General Flag Word	67
4.3	Non-Graphical Data Node	68
4.4	Example of Function Block Procedures	69
4.5	Example of Number Representations	71
4.6	X-Y Function Block	76
4.7	Variable Size X-Y Function Block	76
4.8	Variable Stage Delay	77
4.9	Delay Blocks	78
4.10	Storage of Delay Block Variables	78
4.11	Typical Input Scheme	80
4.12	(a) Analog Input (b) Analog Output	80

4.13	Typical Output Scheme	80
4.14	Digital Input Blocks	82
4.15	Digital Output Blocks	82
5.1	Example of Interconnection of Blocks	87
5.2	Square Root From Square Function	87
5.3	Delay Block Chain	88
5.4	Storage For Variables	89
5.5	Effect of Composite Blocks on RDS (Simplified)	92
5.6	Generation of Control Program	94
5.7	Storage of Variables - Effect on Data Structure	97
5.8	Run-time Values Table	98
5.9	Simple and Composite Block Nodes	100
5.10	Simple and Composite Block Type Nodes	100
5.11	Calling of Function Procedures Via SWITCH	104
5.12	Composite Block Inputs	106
5.13	Main Components of Run-time Data Structure	107
5.14	CORAL 66 Macro Definitions	109
5.15	Function and Terminal Names	112
5.16	Block Name and Engineering Units	113
5.17	Format of RDS Output by GPCL Compiler	114
6.1	Example of Sequencing Problem	119
6.2	Intermediate Stages of Sequencing	119
6.3	(a) LOG Table (b) ALARM Table	123
A.1	Graphical Details of Analog Input Block	133
A.2	Graphical Details of Cascade Controller Block	135
A.3	Graphical Data Structure Example (Part)	138
A.4	Run-time Data Structure Example (Part)	139
A.5	Data Module	140
A.6	P-I-D Routine	140

A.7 Analog Input Routines 141

A.8 Counter Input Routines 142

A.9 Analog Output Routines 143

A.10 X-Y Function Routine 144

A.11 Multiplier Routine 144

List of Tables

7-1 Individual Graphics Storage Requirements 126

7-2 Total Graphics Storage For Block Diagram 126

7-3 Run-time Data Storage Requirements 128

7-4 990/10 Procedure Sizes and Execution Times 128

CHAPTER 1

PROCESS CONTROL

1.1 Introduction

The term 'digital process control' encompasses a wide range of process control functions, all sharing the common factor of involving a digital computer element. The tasks a computer may perform may be divided into 4 levels [1,2] :-

(i) Direct Digital Control (DDC)

(ii) supervisory control

(iii) optimising and adaptive control

(iv) management information .

The areas of interest in this research were DDC and supervisory control.

As its name implies, in DDC the computer performs the actual low level functions of acquiring measurements from the process and adjusting the process actuators. Two factors to be considered when contemplating the use of DDC are the process time constants and the execution speed of the computer. Fast processes which require responses in the order of milliseconds are normally beyond the capability of DDC. A large proportion of processes, however, have time constants of the order of seconds -- it is sufficient to regulate the process actuators several times a second, and DDC is feasible in such cases. Indeed, for such slow processes it is practical to control several loops with one computer.

Since the introduction of the minicomputer in the mid-60's, and especially the establishment of microprocessors by the mid-70's, the trend of DDC has been away from a single big computer controlling several

hundred loops, towards small stand-alone (dedicated) computers handling a small number of loops, because of the increased reliability and reduced costs afforded by the latter [3,4].

Supervisory control, also called setpoint control, is a less frequent control action than DDC, and may also be undertaken by the stand-alone process computers.

In a survey published in 1977 [5] on digital control applications, it was found that applications were most widely reported in the chemical industry (which involve 'slow' processes). A combination of DDC and supervisory control was used in more than half of all the cases reported, and the number of DDC loops varied from the range 1-5 to 100-800. In the majority of applications minicomputers were used, with memory sizes ranging from 8K to 128K words (the majority falling between 24K and 48K). Microprocessor-based controllers have since become more common [4,6-9].

Assuming the technical feasibility of DDC, its implementation involves two main aspects :-

(i) hardware

(ii) design and software .

The viability of most process control systems is linked to economic factors [10]. Hardware costs are incurred by every installation, whereas design and software costs may sometimes be shared among several installations.

Dedicated process controllers may be classified as either special-purpose or general-purpose. Special-purpose controllers are used in two situations. One is where the application is very specialised and stringent physical hardware constraints exist (for example, constraints on execution time or size or weight). The other is in high-volume applications where the hardware costs must be minimised (for example, a washing machine controller). Design costs (for both hardware as well as

software) will consequently be higher, but these are non-recurrent and become insignificant when distributed over the large number of units produced.

General-purpose controllers may be used in a variety of applications. For process controller manufacturers, cost performance is optimised by increasing the number of possible applications. For the end user, cost performance is determined by both the hardware and the implementation effort. The consequence of these requirements is a call for modularised hardware and standardised functions and signals [11]. The effects of modularisation and standardisation are increased hardware costs (certain features are inevitably unnecessary for each specific application) but reduced implementation costs.

The implementation effort may be split into hardware and software design. Hardware design is reduced to the selection of appropriate standardised modules. Hardware standardisation also leads to a reduced programming effort as standard software modules are available to handle the hardware. The remaining software problem is one of defining the collective operation of the various hardware elements to effect the required control functions. This is a non-trivial problem and software costs normally account for a significant proportion of the total system cost. Furthermore, software costs are often recurrent -- programs have to be subsequently modified to suit the occasional change in control requirments.

Software costs will be reduced if some form of standardisation is adopted [12-14]. Normally this is realised through the use of process control-oriented high level languages and a modular approach to software design.

1.2 Programming Languages

Programming languages for process control applications fall into 4 main categories :-

(i) assembly languages

(ii) general purpose procedural high-level languages

(iii) problem-oriented procedural high-level languages

(iv) 'format-defined' languages .

The disadvantages of programming in assembly language are :-

(i) it requires a skilled knowledge of assembly language programming;

(ii) the large amount of code involved necessitates a long software development time;

(iii) errors are easily made;

(iv) errors are both hard to locate and hard to correct;

(v) programs are hard to maintain;

(vi) a large amount of documentation is necessary to describe the actions performed by each section of code;

(vii) the programs, being written in the assembly language for a particular computer, are not transferable to a different make.

These factors result in increasing the software cost.

The only advantage of assembly language programming is the execution and storage efficiency that may be achieved by a highly competent programmer. Assembly language is often necessary to satisfy the constraints imposed upon special purpose controllers, for example where a large amount of processing [15] or high sampling rates are involved.

High-level languages (HLLs) are by comparison much easier to write, read, debug and maintain. In HLLs each statement translates into tens or hundreds of machine-code instructions. Complex programs may be written with relatively few high level language statements. HLLs also have the advantage of portability -- the same programs may be executed on a variety of computers provided that the compilers or interpreters exist. Unfortunately they also tend to be less efficient in terms of execution speed and storage requirements compared to a well-written assembly language program.

Procedural HLLs are those in which the solution is expressed in sequential language statements. They may be subdivided into general-purpose and problem-oriented languages. The former have been designed for general problem-solving whereas the latter are purpose-built to suit their own narrow sphere of application.

The most widely used general purpose HLL has been FORTRAN [5,16]. Standard versions of FORTRAN [17,18] are not particularly suited to process control, so extended versions have been produced [19-21]. Even with extensions, FORTRAN based languages have had only partial success [22], being most successful where a high degree of computation is involved [23].

Other general-purpose procedural HLLs include PL/I [24,25], ALGOL 60 [26], CORAL 66 [27,28], RTL/2 [29,30], and PEARL [31], the last three having been designed for 'real-time' applications. Other real-time languages are listed in surveys by Thompson [16] and Elzer and Roessler [32]. In the U.K., CORAL 66 is most widely used [16].

Due to the general-purpose nature of these HLLs, language features are not specifically geared to any particular application. The realisation of the required control functions still involves a substantial programming effort.

Process control (problem-) oriented procedural HLLs enable the engineer to write programs using familiar engineering terms, and each have their specific area of application. A large number of such languages exist, including AUTRAN [33], PEL [34,35], PROSEL [36,37].

'Format-defined' languages [14] are also problem-oriented HLLs, and are normally of the fill-in-the-blanks type. The user is required to simply fill in forms to describe the control strategy he wishes to implement, the control strategy normally being represented by some diagram readily understood by the engineer. No programming in the normal sense of the word is required. Examples of such languages include PROSPRO [1,38], BICEPS [1], FOX2-IMPAC [36] and ACCOL [39].

A problem with procedural HLLs is the enforcement of programming discipline. This is especially true of general-purpose HLLs, due to their inherent flexibility. Within an establishment, programming conventions have to be standardised and adhered to if confusion is to be avoided.

Format-defined languages, being oriented towards their specific field of application, do not suffer from this problem.

Graphics is a useful tool in process control. Use has been mostly limited to display functions such as the display of mimic diagrams showing the values of relevant process variables [40]. Diagrams of some form are invariably used to express process control schemes. Normally they serve as the starting point in the software generation process -- to assist in the process of writing programs or form-filling.

In certain specific applications a facility is provided to describe the software in a direct graphical way, as in the case of some programmable logic controllers which are programmed interactively using a ladder diagram representation [34].

A graphical (diagrammatic) language is the ideal method of generating DDC software, but little work has been directed towards this goal [41,42]. Kossiakoff and Sleight [43] have described a graphical programming system which utilises 'Data Flow Circuits'.

1.3 Research Objectives

The objective of this research was to design a graphical programming language for programming a standard DDC process control system. This aimed to allow the process engineer to express his control problem in terms of his block diagrams, and also to obviate any necessity for writing programs or form-filling.

The main features of this graphical process control language are :-

(i) complete specification of the control problem using block diagrams in which each block represents a basic control function;

(ii) the facility to define a new set of functions to suit the particular area of application;

(iii) the facility to define groups of blocks as macros or subpictures which may subsequently be identified by single blocks. This further allows segmentation of the diagram into a group of subpictures.

(iv) the provision for commentary text and plant symbols in the block diagrams.

Further processing of the control diagram to generate the run-time system is achieved by a graphic compiler. The main functions of the compiler are the generation of CORAL 66 code and data structure for the run-time system. The latter function includes :-

(i) setting up the data and linkages to define the block diagram structure;

(ii) setting up the data for use by the blocks;

(iii) determination of processing order for the blocks.

The final run-time system is intended to accomodate the following features :-

(i) periodic data-driven execution of the control functions;

(ii) operator interaction which includes the ability to examine and adjust all variables and the ability to reconfigure the control structure;

(iii) logging and display operations.

The emphasis of this approach has been to concentrate on the top level of program design; machine dependent aspects and the implementation of hardware dependent functions can be accomodated as necessary by specialised code modules.

In the design of the language, emphasis has been placed on the data structures to support the graphics and run-time phases. The Graphic Data Structure is designed to facilitate the creation and editing of process control diagrams; in the Run-time Data Structure the emphasis is on execution efficiency and modifiability.

It has not been possible within the duration of this research to produce a complete software system. So far, a graphic editor has been produced to provide the basic editing capabilities. For the run-time system, a supervisor has been assembled capable of performing the most fundamental task of periodically processing the control algorithms. This development has been sufficient to enable the essential features of the software system to be defined, so that the extension to a complete facility becomes a routine programming function.

The essential contribution of this work has therefore been the definition of the software structure required to give efficient implementation of control algorithms based on a graphical language using block diagrams.

CHAPTER 2

SOFTWARE REQUIREMENTS

2.1 Introduction

This chapter gives a general appraisal of the features of a software system appropriate to the requirements of DDC. These features are evaluated to indicate the role of a graphical language and to define a specification for subsequent developments.

The selection of any language for process control is based on several criteria. Criteria related to programming are :-

(i) ease of use

(ii) flexibility

(iii) promotion of structured programming practice

(iv) facility of documentation and maintenance

(v) portability .

Criteria related to execution are : -

(i) efficiency

(ii) system interrogation and modification

(iii) security .

All the languages and systems mentioned in Chapter 1 succeed to varying degrees in attempting to satisfy these requirements.

2.1.1 Ease of Use

The programming task must be undertaken by process engineers, not 'general-purpose' programmers. It has been found that a communication gap exists between process engineers and programmers. The process engineer

knows what he wishes to achieve, but is seldom well-versed in the intricacies of computer programming; the programmer may be skilled in writing programs but seldom understands the requirements of the process control problem [44].

It is therefore preferable that the process engineer be responsible for producing the programs [45,46]. This calls for maximum simplification of the programming task so that he may achieve his control objectives 'without undue effort spent in learning computerese' [47]. At the same time, to assist in the production of working software, the programming language or system must aim to minimise the opportunity for errors.

It has been noted that problem-oriented languages tend to be more suitable for more specific applications than general-purpose high-level languages [22,23,48], as they require a smaller translation effort on the part of the user to convert his ideas into programs. The greater the translation effort required the less satisfactory the language becomes. The engineer is called upon to spend large amounts of time programming; the frequency of errors is related to the ease of the translation process, and the work will become limited to those more qualified 'engineer-programmers'. The translation effort therefore has to be minimised.

2.1.2 Flexibility

The flexibility of a language allows it to be used in a wide range of applications, but this is sometimes achieved at the expense of simplicity. The user may be forced to consider a wider range of options than necessary for the solution of his specific problem of limited scope. On the other hand a problem-oriented language that provides too many

powerful facilities peculiar to a specific application loses its generality and is useless for other applications. It is both impossible and undesirable to attempt to provide all the facilities that are useful to everybody -- undesirable because only a subset will be used for a particular application and the existence of the rest is an unnecessary and unwanted overhead and detracts from the simplicity of the system.

The language should therefore cater for the more common requirements, yet allow the user to modify it to suit his needs. One solution to this is a modular approach which allows features to be added within a defined general structure.

2.1.3 Promotion of Structured Programming Practice

Programs written with a well-defined structure are easier to understand and debug. Languages differ in their amenability towards structured programming, assembly languages being poorest in this respect. Languages derived from FORTRAN and BASIC are also lacking in structured programming aids. The block-structured languages like ALGOL 60, ALGOL 68 [49,50], PL/I and CORAL 66 etc., enable structured programs to be written more easily. The question is not the possibility of writing structured programs (any language may be used with sufficient resolve on the part of the programmer) but the ease with which this may be accomplished.

'Format-defined' languages (which are mainly of the fill-in-the-blanks type) are superior in this respect because the structuring is intrinsic in the facilities provided.

2.1.4 Facility of Documentation and Maintenance

Although documentation plays no direct part in program execution, it is a very important factor in software production, as it directly affects program debugging and maintenance. Maintenance involves two functions :-

(i) correction of 'bugs' discovered after the initial testing period;

(ii) program modification necessitated by the changes in the control strategy and in the process itself [51].

In assembly language programs, the assembler statements do not indicate the overall purpose of a piece of code. It is necessary to include profuse comments, without which it becomes almost impossible to comprehend.

Procedural high-level languages reduce this problem by using more natural syntaxes and allowing the use of meaningful names for variables. Procedural problem-oriented languages allow the user to express the solution to his problem in even more familiar terms, thereby reducing the amount of documentation necessary. In fill-in-the-blanks languages the action of form-filling has the effect of providing automatic documentation to a certain degree.

2.1.5 Portability

A general purpose process control language must be capable of operating on as many different computers as possible. Assembly language programs are not portable -- each range of computers from a manufacturer use their own unique assembly language. High-level language programs achieve portability (although seldom fully) by relying on a compiler or interpreter to translate the language statements into the machine code

for the specific computer. The degree of portability depends on the number of translators available for different computers, as well as the extent to which the various implementations of the language differ.

2.1.6 Efficiency

There are 3 aspects to the subject of efficiency. The first is the programming efficiency and has been discussed under different headings in the preceding sections. The other two aspects are related to program execution : the amount of machine code produced and the time taken for the machine code to execute. As noted in Chapter 1, high-level languages are less efficient in this context. Interpretive languages are generally least efficient. Depending on the implementation, the presence of source text in the run-time system may lead to increased memory requirements; in some systems the necessity to search for program lines and variables, in addition to the necessity to interpret the source text, significantly increases execution time.

Efficiency is less important when the computer available is large and the problem is not a time-critical one; however, process control computers are normally minicomputers or microprocessor-based, especially in the case of DDC.

The program and data structure also affect the overall execution efficiency of a program.

The loss of efficiency caused by a high-level language is often outweighed by the increased efficiency of program generation. The objective is therefore to use a high-level language and to seek efficiency by a careful choice of program and data structure.

2.1.7 System Interrogation and Modification

A flexible process control system should allow examination and modification of all variables. Access to all variables is necessary both during the initial debugging phase, and during subsequent execution, to monitor the proper operation of the system. Due to the impermanence of control requirements and of the process itself, there is often a necessity to modify the control system : introducing additional control calculations, adding variables to be scanned, adding or deleting control loops. All process parameters and variables must therefore be accessible to the engineer or operator [51].

Interpretive languages [46,52,53] allow on-line modification of both program and data. Variables may be accessed through the facility of 'software probes' [53]. Compiled languages require the adoption of a special software structure in which all variables and parameters must be stored in special tables; an interpretive program then executes the various control actions according to the data -- that is, a data-driven system [51,54].

Such a software structure is intrinsic in a graphical block-diagram based system [42].

2.1.8 Security

The process control system must include protection mechanisms to safeguard the systems from being recklessly or mistakenly modified with catastrophic results. There are two aspects to the problem of security. The first is to do with the protection against unauthorised access -- normally achieved by the use of passwords -- but is not of concern here. The second aspect of security involves the minimisation of the effects of human errors.

In procedural interpretive languages (for example BASIC and its derivatives) new program lines may be checked for syntactic errors before being accepted. This does not prevent logical errors such as the creation of endless loops or the illegal accessing of variables.

Data-driven systems are more secure due to the limited modification allowed, assuming the correct operation of the basic control routines.

2.2 Definition of the Graphical Process Control System

A system is now proposed with the objective of embracing the requirements described in the preceding sections. It consists of three parts :-

(i) the Graphical Process Control Language (GPCL) ;

(ii) a program development system comprising a graphics editor and the GPCL compiler;

(iii) the run-time system which is generated for execution in a process computer.

The following sections define the requirements of this software system in relation to the factors discussed in Section 2.1 .

2.2.1 Ease of Use

Since process engineers use block diagrams as their standard way of representing ideas and control schemes, a language based on these block diagrams has considerable advantages to the user. Several languages are based on this concept, but still require the user to specify textually the contents of his diagram [6,55-57].

Ideally the program should be identified with the diagram itself -- that is, to allow the engineer to input his process control diagram into the computer and let the computer generate the control software. The human effort then becomes one of transcription rather than translation, and is therefore susceptible only to transcription errors which are less common than translation errors.

With the use of sophisticated pattern recognition techniques and an imaging device (e.g. a TV camera) to read the user's drawing the whole process may be automated, thus eliminating even transcription errors. At present however an economic solution is to require the user to generate his diagram on a graphics visual display terminal.

A graphical, block diagram language is therefore specified, with a full graphics support facility. This language is called GPCL (Graphical Process Control Language).

2.2.2 Flexibility

Rather than attempting to provide all the blocks that may conceivably be required, it is essential to provide the facility to define new blocks whenever required for the particular application. The implementation envisaged here has identified the essential features of blocks to ensure that structural requirements can be met for any block likely to be needed.

The further introduction of a 'macro block' as a means of replacing a group of blocks by a single block adds to the flexibility of the language.

2.2.3 Promotion of Structured Programming Practice

Two levels of programming are involved in GPCL. The basic level is the programming to realise the control action of each block -- this is only required when new blocks are introduced. The GPCL language defines a standard convention for handling data (Chapters 4,5).

The other level is the 'programming' of the diagram, which is in actuality the interactive creation of the diagram at the graphics terminal. For this, the 'macro' facility enables 'top-down design' by allowing the structuring of the diagram into blocks of different levels.

2.2.4 Facility of Documentation and Maintenance

The method described is inherently self-documenting. In fact, since no translation (from diagram to program) by the user is required, the need for documentation beyond the level of overall system description does not arise.

Since the 'program' is equivalent to the process control diagram itself, there is no problem of comprehensibility. Maintenance is therefore facilitated.

2.2.5 Portability

In GPCL, the question of portability occurs twice -- the portability of the GPCL editor and compiler and portability of its output. To minimize machine dependence the GPCL compiler and editor must themselves be written in a high-level language. The portability of the generated software (the GPCL compiler output) would be lost if it was a specific machine dependent code. It is preferable to produce an intermediate language output, which can then be converted into the

machine language for the target process computer (the software development stage need not be supported by the process computer).

While it is possible to design a new purpose-built intermediate language from which machine specific code may subsequently be generated, this was not done because of the significant extra effort required and the necessity to provide code generators for different target computers. The choice was therefore made among several high-level language alternatives.

The application under consideration being process control, the language had to be one of the 'real-time' langauges. The popularity of CORAL 66 for real-time applications and the availability of a CORAL 66 compiler for the Department's Texas Intstruments 990/10 minicomputer, prompted the adoption of CORAL 66 as the intermediate language. The choice of CORAL 66 as the intermediate language led to its use in the entire software system -- the GPCL editor and compiler.

2.2.6 Efficiency

Although the code compiled from a high-level language is generally not as efficient as good assembly code, CORAL 66, having been designed with a consideration to real-time requirements [27] is relatively efficient. This by itself is not sufficient -- the control program generated by the GPCL compiler has to be designed to be as efficient as possible. This is mainly achieved by eliminating the necessity for any searching during execution of the control algorithms.

2.2.7 System Interrogation and Modification

It is natural to implement a block-diagram based language as a set of tables which are operated upon by an interpretive program. The use of tables allow both variables and control strategy to be examined and modified.

Further, the GPCL language allows text such as names and engineering units to be attached to each block to facilitate identification.

2.2.8 Security

The execution of the software generated by the GPCL compiler is data-driven. As noted in Section 2.1.8, this makes it less vulnerable to user-induced catastrophes, since the user is not permitted to modify control routines.

2.3 Software Generation Process

Several steps are involved in the process of generating the final control program for use in the process computer (Fig 2.1) :-

(i) synthesis of diagram using GPCL editor;

(ii) compilation by the GPCL compiler to produce the control program in CORAL 66 form;

(iii) compilation by a CORAL 66 compiler to produce the control program in machine-dependent code form;

(iv) linking with any machine dependent routines;

(v) loading into the process computer;

(vi) execution.

Steps (i), (ii) and (vi) will be discussed in the following chapters.

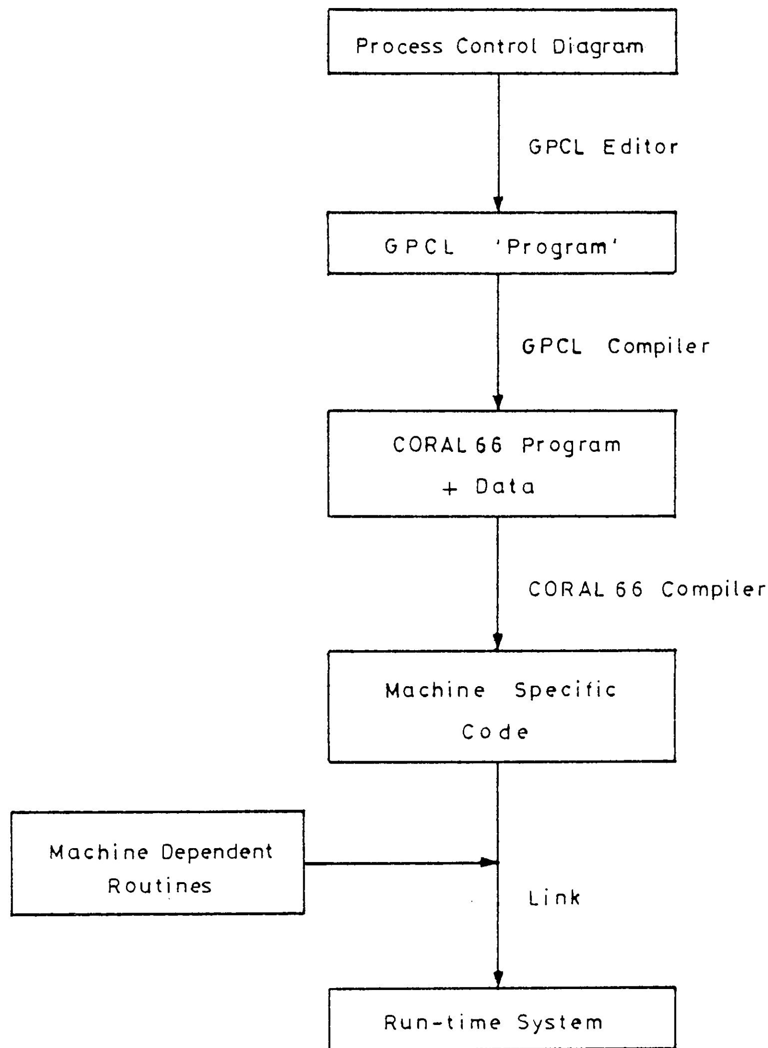


Figure 2.1 Software Generation Process

In many DDC applications the process computer may be microprocessor-based, with a small memory capacity. This implies that the software generation process must be accomplished on a different computer -- a minicomputer and graphics terminal is needed.

CHAPTER 3

GRAPHICS

3.1 Introduction

Graphics being a natural medium for man-machine communications, its applications are varied and too numerous to mention. Some areas where it is extensively used include simulation [58-61], computer aided design [62-64], analysis [65,66], and computer animation [67,68]. Whatever the application, two basic components are involved -- graphics software and hardware.

3.1.1 Graphics Software

Graphics software may be grouped under 2 headings -- application programs and graphical languages. Application programs are user-written programs which perform the computations peculiar to his problem. The application program makes use of the graphical language to perform its graphical input/output actions.

Normally the graphical language is based on an existing language [69] (normally the same one as used by the application program) -- it may either take the form of a graphic package (a set of graphical functions implemented as subroutines to be called by the application program) [70-76], or take the form of extensions to the language [77,78]. The graphical language includes facilities for geometric transformations (scaling, rotation, translation) as well as simple line-drawing, and may often handle 3-dimensional pictures.

An important component of graphics programs is the data structure. The data structure is the software representation of the model being operated upon. 'The purpose of the data structure is to facilitate the extraction of intelligence and manipulation of both the image and the information it represents' ([Abrams [79]). It must store graphical information in such a way that it can be retrieved and manipulated easily. It must also reflect the non-graphical characteristics of the model.

Much work has been done on the subject of data structures. Sutherland in SKETCHPAD [80] describes a ring structure for handling a common class of pictures which may be termed 'network graphs' -- configurations in which pictures are connected to others in a network, the pictures often being decomposable into lower level 'subpictures'. Others have discussed the design of general-purpose and tailored graphic data structures [79], data structures for remote computer graphics [81], data and storage structures [82], a simple data structure for drafting [83] and line drawing [84], and data structures for picture processing [85]. The judicious choice of data structure is vital to the success of a graphics program.

3.1.2 Graphics Hardware

A variety of graphics devices exist. They are of 2 types -- interactive or passive. Passive devices are output-only devices, like the graph plotter. Interactive devices (graphics terminals) permit human interaction through a variety of input mechanisms.

Graphical input devices include the keyboard, switch, joystick, light pen, and data tablet [86,87]. Each of these devices provide a different means of expression for the user. Some are more suited to the

expression of values, or text, or selection of objects. Each input device requires different software to operate (device drivers). In order to achieve some degree of portability for the graphical language, they may be categorised according to their functional characteristics. Several classifications have been suggested [87-91]. Foley and Wallace [89] propose 4 device types -- pick, button, locator and valuator. Examples of these are the light-pen (pick), function key (button), joystick (locator) and potentiometer or analog dial (valuator).

Despite the functional differences, it has been attempted to demonstrate that each device may be treated as belonging to any of the four device types, and 'with appropriate programming, any device can simulate any other device' (Cotton [88]), although in some cases the simulation may be quite awkward.

Graphic displays (terminals) may be grouped into 3 main types [87,92] -- refreshed directed beam CRT, raster-refreshed CRT, and direct-view storage tubes. In the refreshed directed beam CRTs [87,93], lines are drawn by directing an electron beam across the screen. Since the image only remains on the screen for a fraction of a second, all the lines making up the picture have to be continually redrawn (refreshed). Lines are drawn in a similar way in direct-view storage displays [94], but the image does not fade and therefore no refreshing is necessary. Raster-refreshed (also called raster-scan) CRTs are similar to television CRTs; they require the generation of a matrix of intensity values which are fed to a TV monitor. The matrix is stored in a 'frame buffer'.

Again the use of different displays require different device drivers. Attempts [73,76,95,96] have been made to achieve device independence by separating the whole graphics software into 3 parts -- an application program, a standard graphics package, and a device driver.

The most basic functional difference between the refreshed directed beam CRT, raster-scan CRT, and the direct-view storage tube is the necessity to refresh the first two, and conversely the ability to modify the image continuously. Storage technology requires the whole screen to be erased and the picture redrawn. Drawing speeds are relatively slow for storage tubes.

Despite the device independence achieved between devices of the same type (even between the two refresh types) the three types are not fully interchangeable. Carlson [92] and Preiss [94] identify the graphics terminal requirements and their related application areas. The basis for selection of a particular type of display depends on several factors, including the ability to dynamically move objects, the resolution available, and not least the cost.

Functionally, both refreshed directed beam and raster-scan CRT displays may be made to simulate storage tube displays, within limits.

3.2 Graphical Process Control Language (GPCL)

As has already been stated in Section 2.2.1, the language proposed is both block-diagram oriented and also allows the process engineer to describe his diagram by graphical means (through a graphics terminal using the Graphic Editor) instead of by a textual language or form-filling.

Fig 3.1 shows an example of a block diagram. It is a diagram of a simple dye mixing process. Blocks 11, 12 and 13 are input blocks; they represent the hardware interface to the 'real world', and obtain their inputs from a densitometer and two flow meters. The control loop applies its output through block 14 to valve V2 in response to changes in flow induced by a change in V1. A complete control system would be more

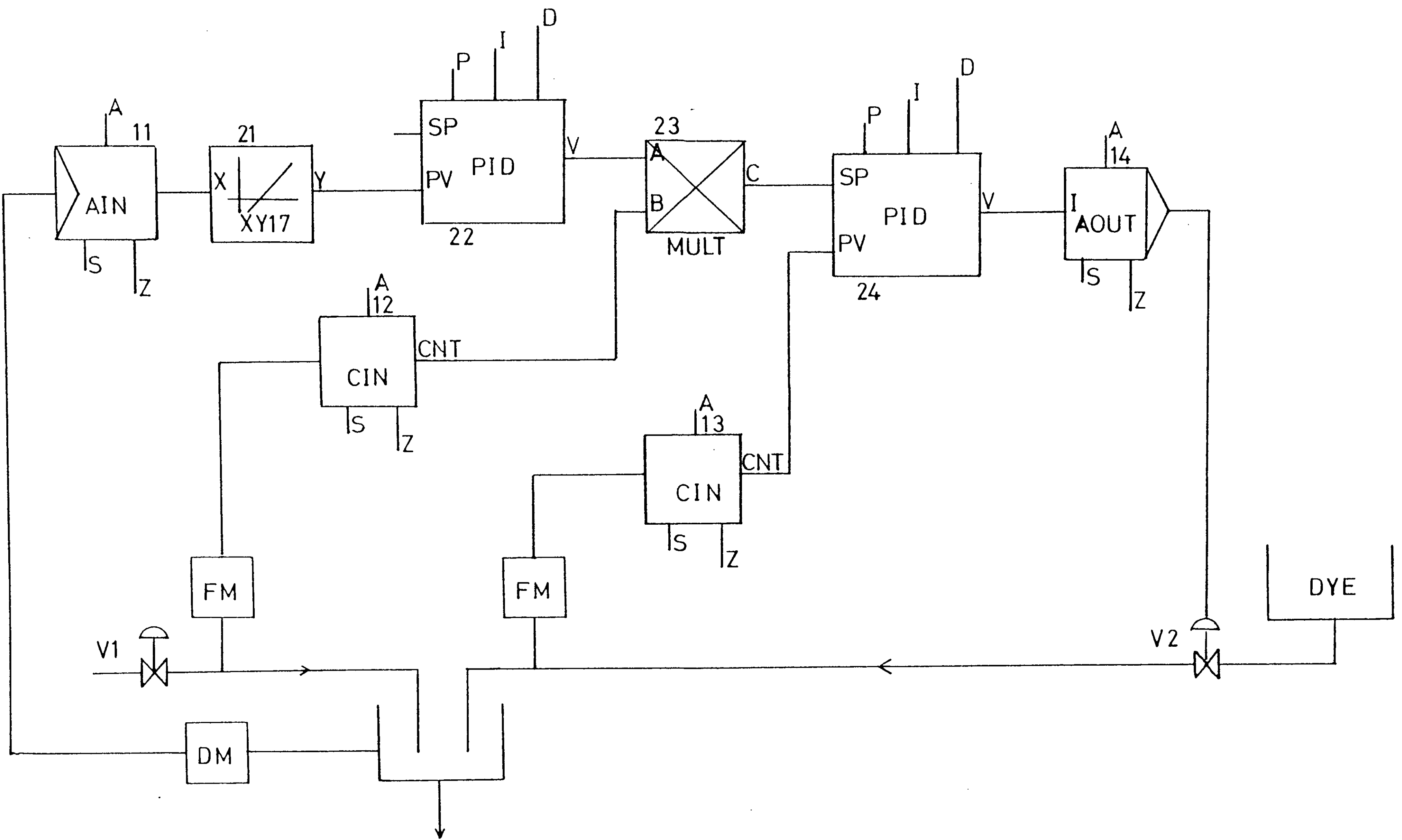


Figure 3.1 Dye Mixing Process Block Diagram

complex as it would require safeguards like level limit detection.

3.2.1 Macro Facility

A feature of GPCL is the facility to form composite blocks by grouping together simple blocks. In Fig 3.1 all blocks are simple blocks -- that is, they are basic components and cannot be decomposed further. If cascade control is often used, the engineer might wish to define a cascade controller block, made up of two PID blocks and a multiplier (Fig 3.2). This is similar in many ways to the 'macro' facility in

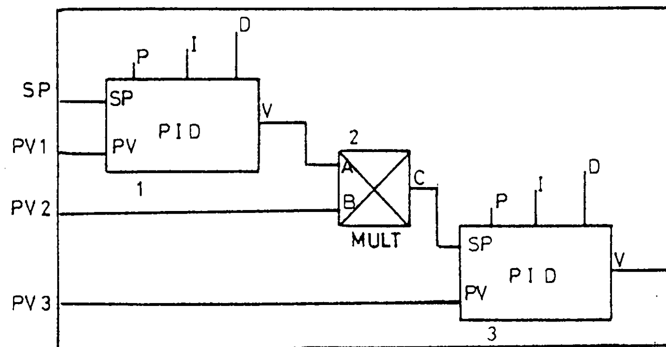


Figure 3.2 Cascade Controller - Example of Composite Block

textual programming languages.

The overall control scheme can then be redrawn as in Fig 3.3 , where block 27 is the composite block.

3.2.2 Components of GPCL

A GPCL 'program' is made up of a graphical component and a non-graphical component. The graphical component is the representation of the block diagram and is called the Graphic Data Structure (Section 3.3). The non-graphical component is formed by the CORAL 66 routines which perform the functions represented by each block, and other functional descriptions which do not manifest themselves in the picture. These will

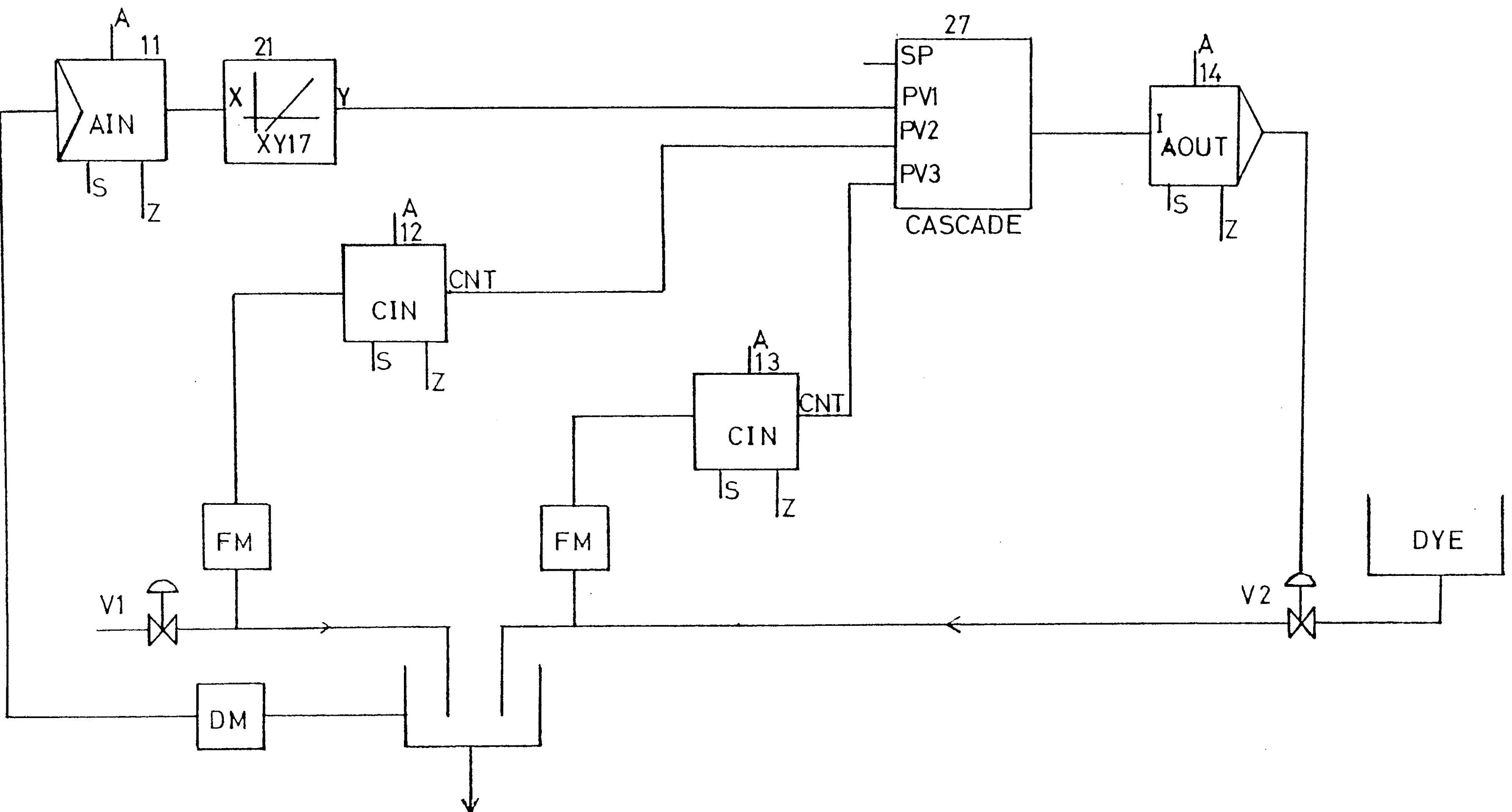


Figure 3.3 Block Diagram With Composite Block

be discussed in Chapter 4.

3.2.3 Main Graphics Requirements

While the graphics facilities required for the Graphic Editor may be very sophisticated, only a basic subset of graphics functions is essential. The possible graphics facilities (either hardware or software) are listed below together with their usefulness in this application.

1) 3-D representation -- this is not needed, since block diagrams are 2-dimensional drawings.

2) Colour -- process control block diagrams are normally monochromatic (plant mimic diagrams are often multi-coloured, but are not relevant here). However, during editing, the availability of colour is advantageous as it can help to emphasise the objects of interest. The objects themselves need not possess a colour attribute in the Graphic Data Structure -- the graphic editor may draw objects with different colours according to the current editing circumstances.

   Colour is not available on direct-view storage displays.

3) Intensification -- this is the ability to display objects at different levels of brightness. This feature is again useful, but not essential, for editing purposes.

4) Translation -- this is a geometric transformation that affects the position of a graphical entity. This is required in order to position blocks at different parts of the screen, and also to enable viewing of different parts of the diagram if it is too big.

5) Rotation and mirroring -- rotation is a transformation that affects the angular orientation of a graphical entity. Arbitrary rotations are not needed since block diagrams should be laid out such that blocks are either 'horizontal' or 'vertical'. Rotation through 90 and

270 degrees would be of some use. However, this significantly increases the complexity of the program in several ways. If the rotation is not available as a hardware facility, it places a burden on the software. In a terminal with hardware character generation but no rotation, the rotation of text through any angle becomes a problem.

Rotation of a block through 180 degrees is almost never wanted as the block ends up upside down. The action required is 'mirroring'. Again text causes a problem since it should not be mirrored as well, and the question arises as to where to position the text associated with the mirrored block.

The absence of rotation and mirroring places a constraint on the block-diagram layout, but greatly simplifies the software and hardware requirements.

6) Scaling -- scaling is a transformation that changes the size of a graphical entity. In many graphical drawing languages [80,81,84, 97,98] 'macros' or 'subpictures' are used to define groups of more elementary graphical entities, and as such, when a 'macro' is encountered during picture drawing, it is expanded and its constituent components drawn instead.

In GPCL, composite blocks retain their graphical identity and are not expanded when a block diagram is drawn. Thus scaling, otherwise required to draw the internal components, is not required.

The question that remains is whether blocks of the same type should be drawn in different sizes, and whether 'zooming' is allowed. 'Zooming', the display of part of a picture at various magnifications is not a useful feature in the context of process control block diagrams. The use of different-sized blocks can provide a means of indicating the relative importance of blocks. The attendant

disadvantage is that of text generation, as in the case of rotation. Extra storage is also needed to store the scale for each block. In GPCL therefore blocks may only have one size.

7) Dynamic display -- editing of a block diagram is facilitated by the ability to continuously move objects about the display area ('dragging'), and also the ability to 'flash' certain blocks and user prompts on and off. This is only possible with refresh displays. While this feature facilitates the editing task, it does not affect the GPCL language in any way.

3.3 Graphic Data Structure

The Graphic Data Structure (GDS) is a data structure which contains the complete graphical information of the block diagrams. It is a hierarchical structure, but differs from other hierarchical structures [79-82,84,97,98] in that the hierarchy pertains to the functional rather than graphical properties. The significance of composite blocks is more functional than graphical.

The Graphic Data Structure also includes a small amount of data of a non-graphical nature. Its presence enables a certain amount of error-checking to be performed at an early stage (during graphic editing).

3.3.1 Design Criteria

The general criteria for graphical data structures are the same -- a need for adequate representation of the problem model, providing sufficient flexibility for the purposes of the application, and facilitating extraction and manipulation of information. At the same time the storage should be efficient (not require too much space). These criteria are generally interrelated and mutually contradictory, and

solutions are always a compromise.

Abrams [79] addresses the problem of choosing between an existing general-purpose graphical data structure, and one that is tailored to the specific application. The former is normally less efficient of storage because of the presence of unused features; the latter requires a construction effort.

In the interests of efficiency it was decided to use a special-purpose data structure.

3.3.2 Attributes of Simple Blocks

In a process control block diagram, several different graphical entities exist -- function blocks, lines, text and plant symbols.

Function blocks (which will sometimes be referred to simply as 'blocks') define the processing that is to be performed, and they may be software analogies of hardware signal processing modules.

Lines are of two types. The first type is analogous to wires that link the hardware modules (function blocks). They define the signal flow amongst the blocks. The second type of lines serve no functional purpose -- they are purely pictorial elements (for example to represent a pipe).

Similarly, plant symbols are purely pictorial elements, that serve to make the diagram more meaningful.

A piece of text may or may not be associated with a particular block. If it is, it may be purely commentary or provide textual identification and information such as names and engineering units, or it may have a numerical value, like a constant for a setpoint.

By treating all the graphical entities as attributes of the BLOCK, the latter may be identified as the fundamental component of a diagram. It should be noted that a BLOCK need not be a FUNCTION BLOCK -- it may

not perform any processing function, in which case it is a pure GRAPHIC BLOCK.

The graphical attributes of a BLOCK are :-

(i) block number

(ii) position

(iii) shape (and size)

(iv) input/output structure

(v) connections

(vi) text .

The block number is unique to each block within the picture being displayed. A 'picture' here refers to a collection of blocks forming a block diagram. A more general definition is given in Section 3.4 .

The position is its x-y coordinates within the picture.

The shape of the block, and its input/output structure (the number of input and output terminals and their positions) are the same for all blocks of the same TYPE. As stated previously, the size is fixed and therefore not explicitly required.

Connections are the lines that define the signal flow, and text is that associated with the particular block.

Not all blocks possess all the above attributes. GRAPHIC BLOCKS do not possess any input/output structure, nor therefore any connections.

By defining the BLOCK as the fundamental component of a diagram, plant symbols and the second type of lines described above may be defined as pure GRAPHIC BLOCKS. Text which is not related to any particular block may be treated as block-related text by attaching them to graphic blocks.

In this treatment lines do not exist as entities in their own right -- they are implicitly defined by connections or graphic blocks. A problem arises when sometimes, for aesthetic or logical reasons, connecting lines between two (function) blocks have to be composed of

several segments. The solution to this is to define a special JUNCTION block, which has one input and one output (in the same position) and has the shape of a big 'dot'. Signals are then routed via these JUNCTION blocks.

A FUNCTION BLOCK also possesses several non-graphical attributes. These include the function algorithm (in the form of a CORAL 66 procedure) and several properties which determine the legality of interconnections (to detect errors such as connection between incompatible terminals). These properties are called Non-Graphical Data and will be discussed in detail in Chapter 4.

3.3.3 Single-level Data Structure

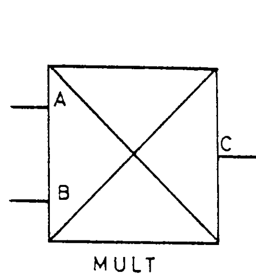
This section deals with a possible method of representation of a picture which consists of only simple blocks.

The input/output structure and shape of a block are type-specific, being identical for all blocks of the same TYPE. In addition, all FUNCTION BLOCKS possess a BLOCK TYPE NAME (FUNCTION NAME) and normally have a name for each of their input and output terminals. Fig 3.4 shows block schematics for the MULTIPLIER and INTEGRATOR blocks. These names are common to all blocks of the same TYPE. GRAPHIC blocks need not possess a block type name.

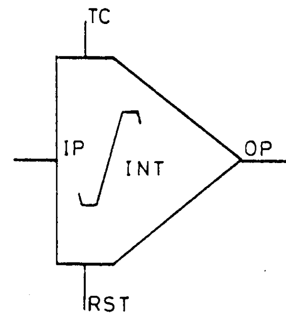
All type-specific information is stored in a table called the Graphic Information Table, which contains a record for each different type of block (Fig 3.5). Each record in the table holds the following information :-

(i) length -- total length of the record

(ii) TYPE number -- unique to each block TYPE



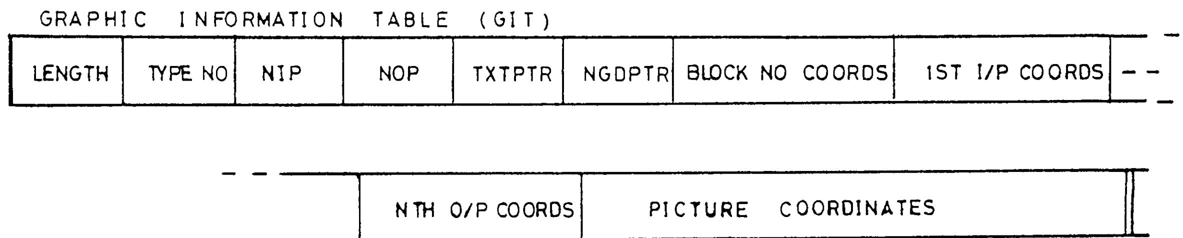
MULTIPLIER



INTEGRATOR

Figure 3.4 Block Schematics

GRAPHIC INFORMATION TABLE (GIT)



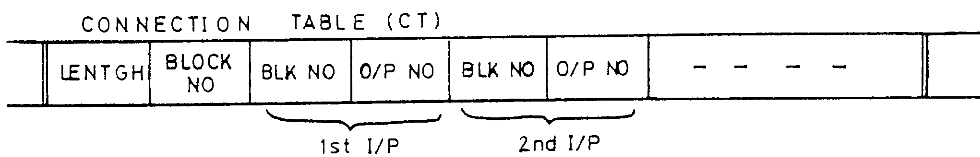
GRAPHIC INFO POINTER TABLE

TYPE	GIT ADDR.

MASTER TABLE

BLOCK NO	TYPE NO	X-Y COORDS	CT ADDRESS	TXPT ADDR

CONNECTION TABLE (CT)



TEXT POINTER TABLE (TXPT)

X-Y COORDS	TXT ADDR	NEXT

TEXT TABLE (TXT)

LENGTH	TEXT

NON-GRAPHICAL DATA TABLE

NIV	GF	ROF	LIF	LOF	CIF

Figure 3.5 Components of Simple Graphic Data Structure

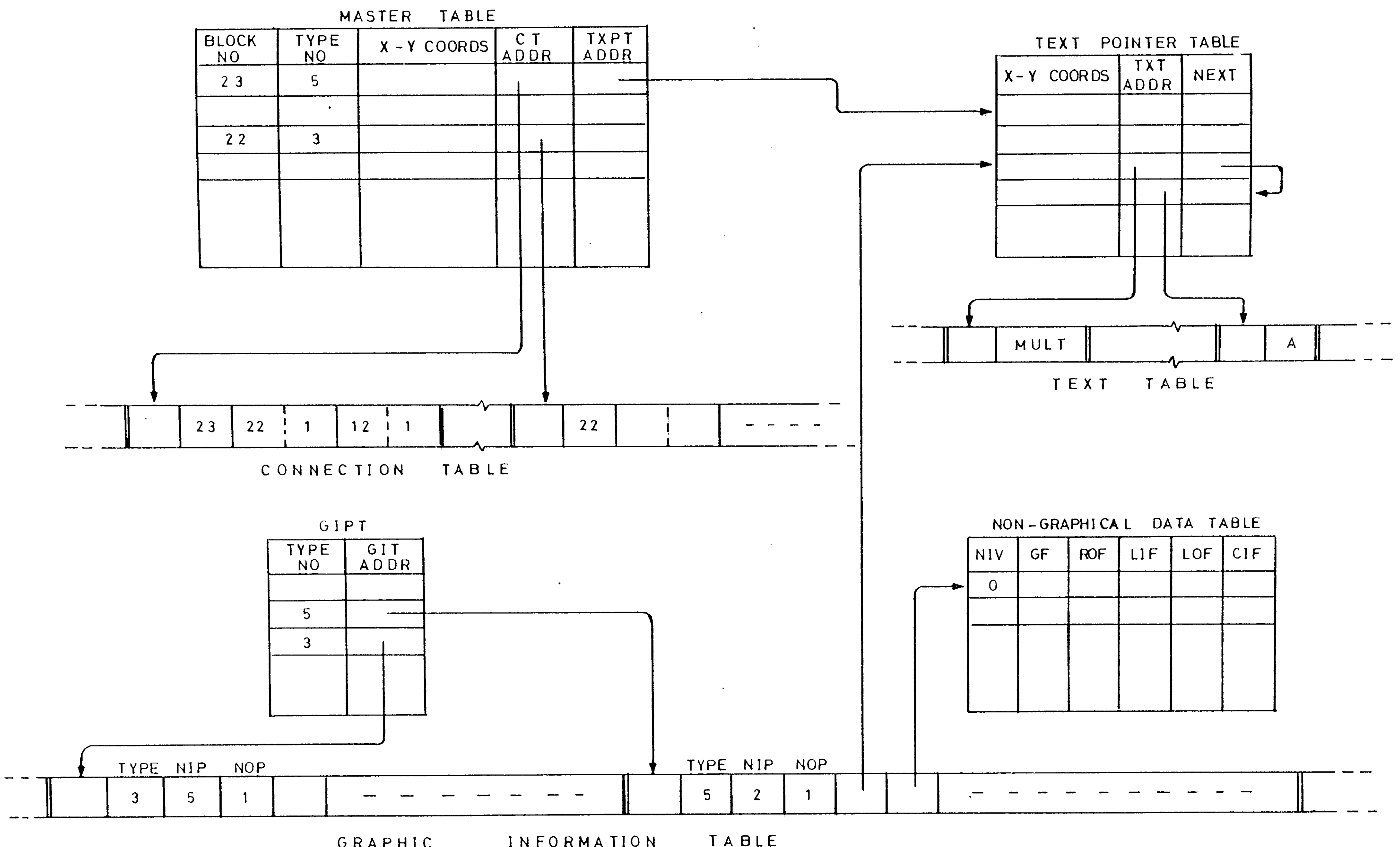


Figure 3.6 Simple Graphic Data Structure

(iii) number of inputs (NIP)

(iv) number of outputs (NOP)

(v) a pointer (TXTPTR) to any type-specific text

(vi) a pointer (NGDPTR) to any non-graphical data

(vii) coordinates of the block number -- all blocks have a block number by which they may be identified. Normally all block numbers will be printed next to the block; this information specifies a suitable place for them to be printed.

(viii) the coordinates of each input and output terminal -- this is needed to draw connections in the correct places, and to allow the terminals to be referred to via a graphic cursor during editing.

(ix) the picture coordinates describing the lines that form the shape of the block.

All coordinates are specified relative to the base of the block which may be any point in or around the block. It is also the position of the block type name if one exists (the <u>position</u> of a piece of text is the bottom left-hand corner of the first character).

The picture coordinates specify the endpoints of the lines, in the order to be drawn. Only one end point is specified for each line -- the start point of the line is the end point of the previous line. 'Moves', or invisible lines, are flagged by adding a big number (2000) to the x-coordinate.

Since both the I/O structure and the shape vary considerably between different types of blocks, the records in the Graphic Information Table are of variable length, the table itself being organised as a linear array. A directory called the Graphic Information Pointer Table points to the start element of each record. The remainder of the

attributes are stored in five other tables -- a Master Table, a Connection Table, a Text Table and Text Pointer Table, and a Non-Graphical Data Table (Fig 3.5).

The Master Table holds for each block a record containing the following pieces of information :-

(i) the block number

(ii) the TYPE number of the block

(iii) the x-y coordinates of the block

(iv) a pointer to the block's connection information in the Connection Table

(v) a pointer to the Text Pointer Table .

The Connection Table is a table containing variable-length records -- one for each block that has an I/O structure. Each record holds the following pieces of data :-

(i) length of the record

(ii) block number

(iii) for each input, the block number and the output to which it is connected.

Since the connection of outputs to inputs is a one-to-many relationship, and since it is not permissible to interconnect two outputs, this is sufficient to define all interconnections.

Two tables are required to hold text information. Each block may include several pieces of text, each being in a different position in relation to the block. The text strings are stored as variable length records in the Text Table. The start of each text record, together with the coordinates of the first character of the text string, is stored in an entry in the Text Pointer Table. Also stored in each entry in the latter table is a link to the next piece of text which belongs to the same block.

Each function block has a record in the Non-Graphical Data Table which stores, among other things, information on the type of signal (continuous or logical, fixed or variable -- Section 4.2) compatible with each terminal. This information is not essential during the editing phase (it is required in the compilation phase) but its presence enables signal checking, and therefore error detection, at an early stage in the software production process.

The result is a Graphic Data Structure as shown in Fig 3.6 . Such a structure suffers from two major deficiencies. It does not allow for the representation of composite blocks, and also does not lend itself to efficient manipulation. These two problems are dealt with in the next section.

3.3.4 Multi-level Structure

The previous section dealt with a single-level data structure -- that is, a picture which consists only of simple blocks. A powerful and useful feature to have in any programming language is the facility for creating 'subroutines' or 'macros'. In the context of GPCL, it means the ability to create a subpicture or 'composite block' which is made up of several other blocks. This is useful for the following reasons :-

(i) it allows for neater, more comprehensible diagrams;

(ii) frequently used configurations can be defined as macro blocks which can be called up when required;

(iii) it encourages a structured, modular approach to designing control systems.

An arbitrary example of composite blocks is shown in Fig 3.7 . A, B, E, F and G are simple block types; C and D are composite block types. Fig 3.8 shows the multi-level nature of the diagram, where the squares

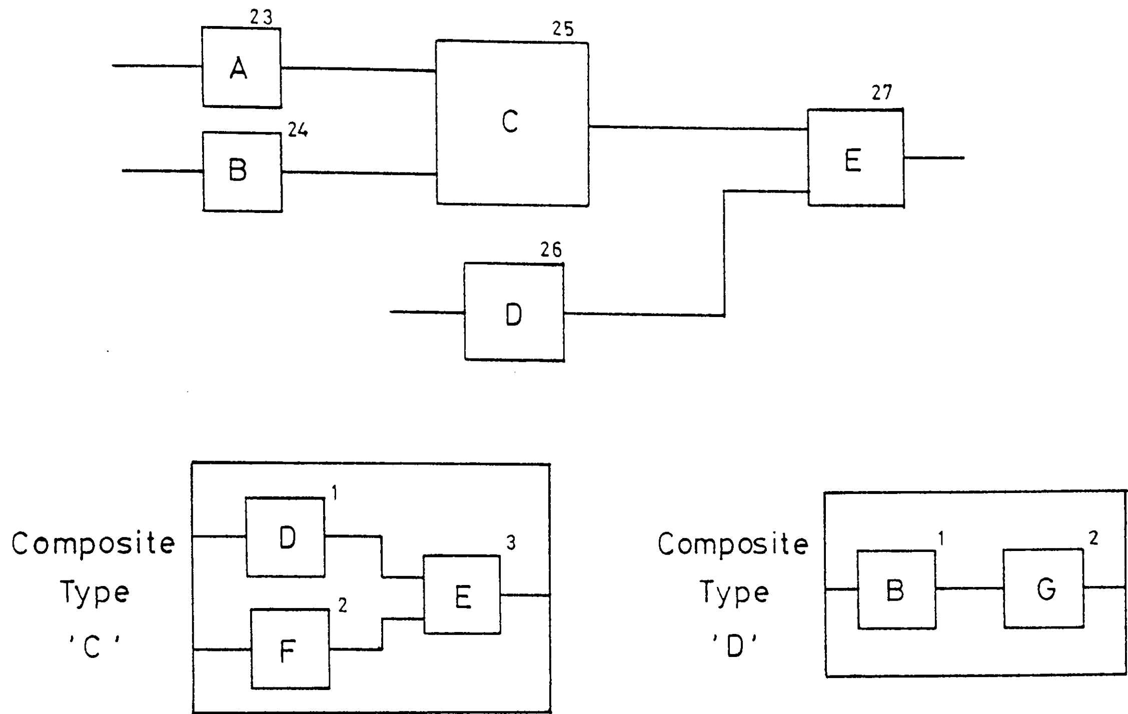


Figure 3.7 Example of Composite Blocks

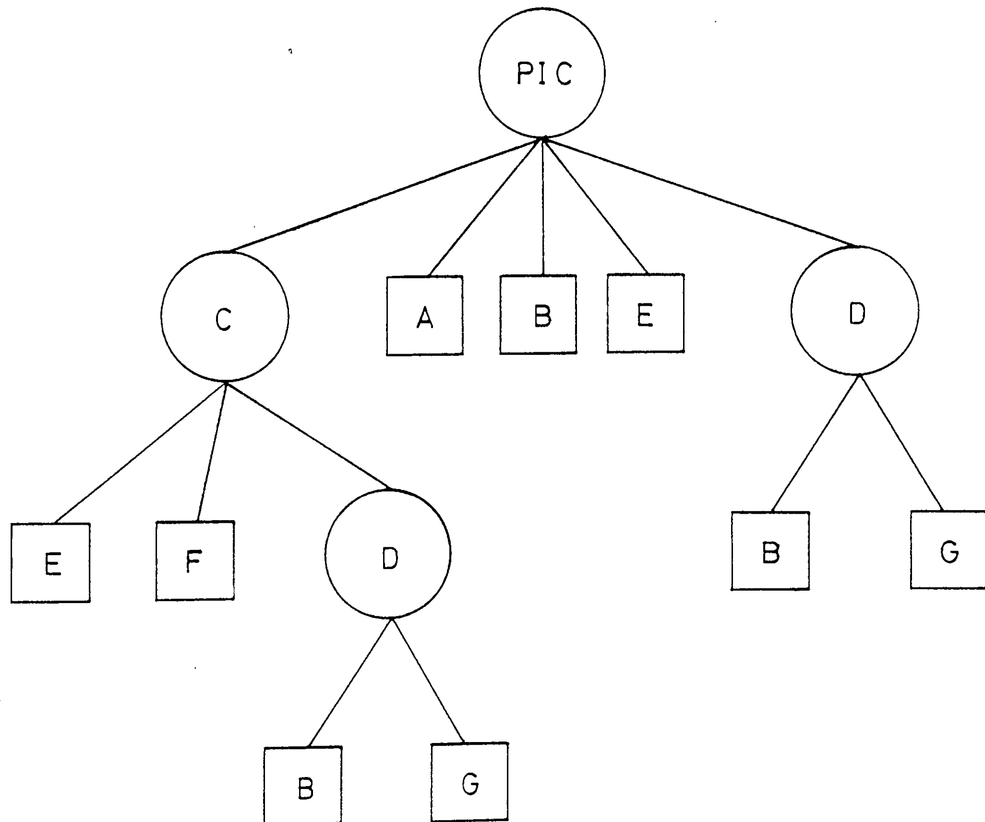


Figure 3.8 Multi-level Nature of Diagram

represent simple blocks and the circles composite blocks. It is clear that each composite block resembles a simple picture and can therefore be represented using a data structure fairly similar to that of the previous section.

Two classes of composite blocks may be identified. The first class is the 'one-off' composite block -- it has been defined not because it is to be used more than once, but for conceptual or aesthetic purposes. In GPCL, this class of composite block is called a SUBPICTURE. The second class of composite block has been defined with the intention of repeated use. In GPCL this is called a MACRO. A main difference between the two classes of composite blocks arise in the run-time system -- the internal configuration of a macro cannot be modified whereas that of a subpicture can. This has the implication that in a process control diagram, each subpicture can appear only once.

The Graphic Data Structure described in the previous section has several shortcomings when operated upon by the computer.

In the Connection Table the input information is given in terms of block numbers. Since there is no direct relationship between a block number and the position of the entry for that block in the Master Table, a search through the latter is required. A search is required for every connection -- a time consuming process as the number of blocks increases.

The same problem arises when accessing the Graphic Information Table. A search has to be performed to access the right record for each block type.

The Connection Table only holds information for inputs. Since a many-to-one relationship exists between inputs and outputs this is most efficient in terms of memory usage. However, it becomes a tedious process to try to determine all the inputs which are connected to a particular output, as every entry in the Connection Table has to be searched. While

the delays are insignificant for a small system, for a larger system with a substantial number of blocks the search time can become intolerable.

The existence of several tables is another potential source of problems. It is not easy to allocate suitable sizes for each of the tables, and there is therefore a danger of overflow. CORAL 66 does not allow for unbounded arrays and tables.

The overflow problem is less likely to occur with a single array. All records are therefore physically stored in the same array, with records belonging to the same table forming a linked list.

In order to minimise searching, TYPE information in the Master Table is converted into actual pointers to the appropriate Graphic Information Table records, thus dispensing with the Graphic Information Pointer Table. Similarly, connection information does not make use of block numbers, but pointers to the relevant Master Table record. Furthermore, connection data includes information for each output -- all input terminals connected to the same output are stored in a linked list. In a linked structure such as this, it is more appropriate to refer to the table records as NODES. Each record in the Graphic Information Table is referred to as a Graphic Information node. For each block, connection information is now merged with its record in the Master Table, forming a Block node. The Text Pointer Table and Text Table are also merged, forming Text nodes. The resultant structure is shown in Figures 3.9 and 3.10 .

All nodes have three common features. The first word is always a link to some other node, the second word always contains the length of the node, and a null link or pointer is indicated by a zero value.

In the Block node, BLINK links all other nodes at the same level. GIPTR points to the Graphic Information node. TXTPTR points to the first related Text node. For each input, BLKPTR points to the Block node of the

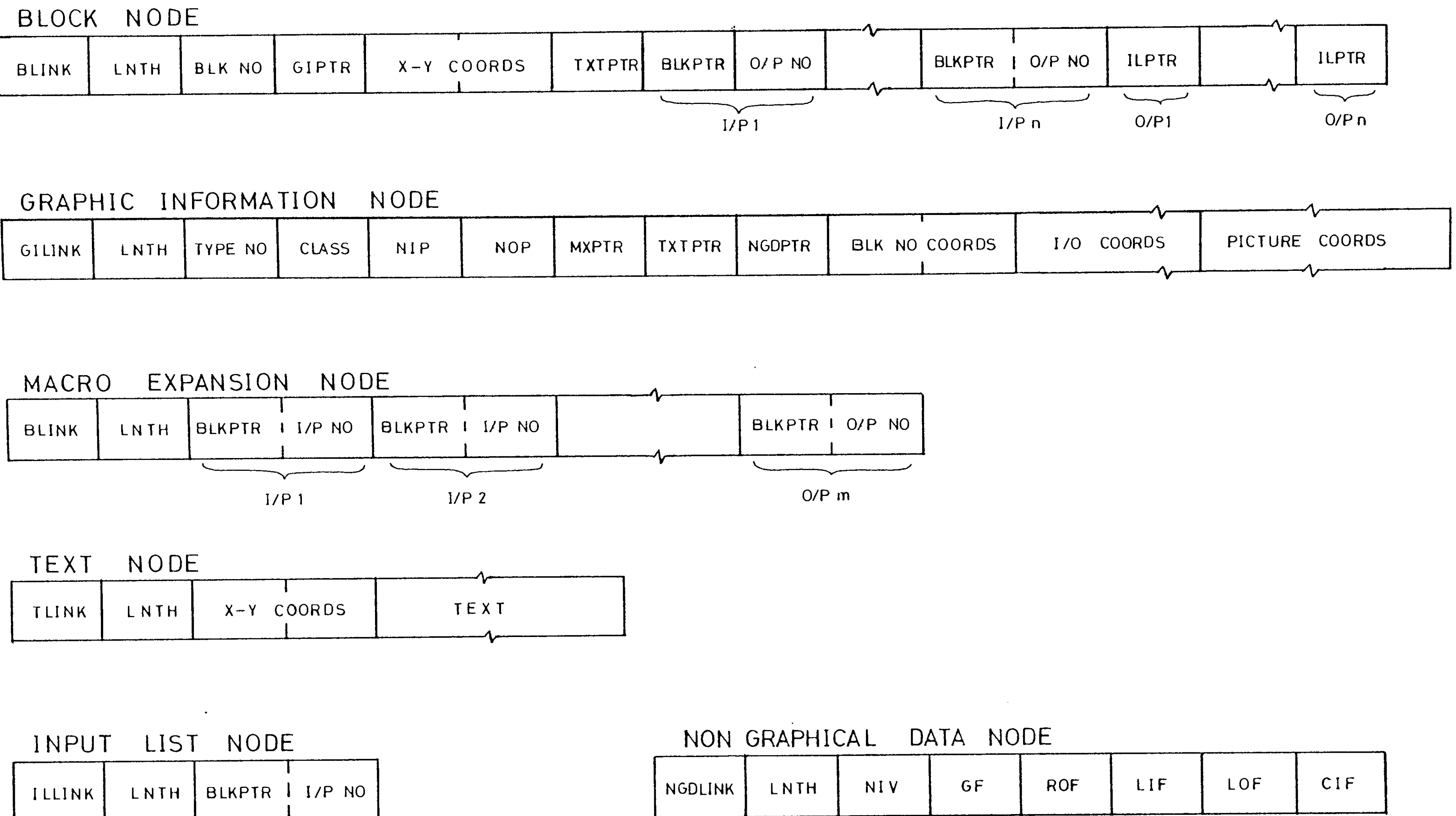
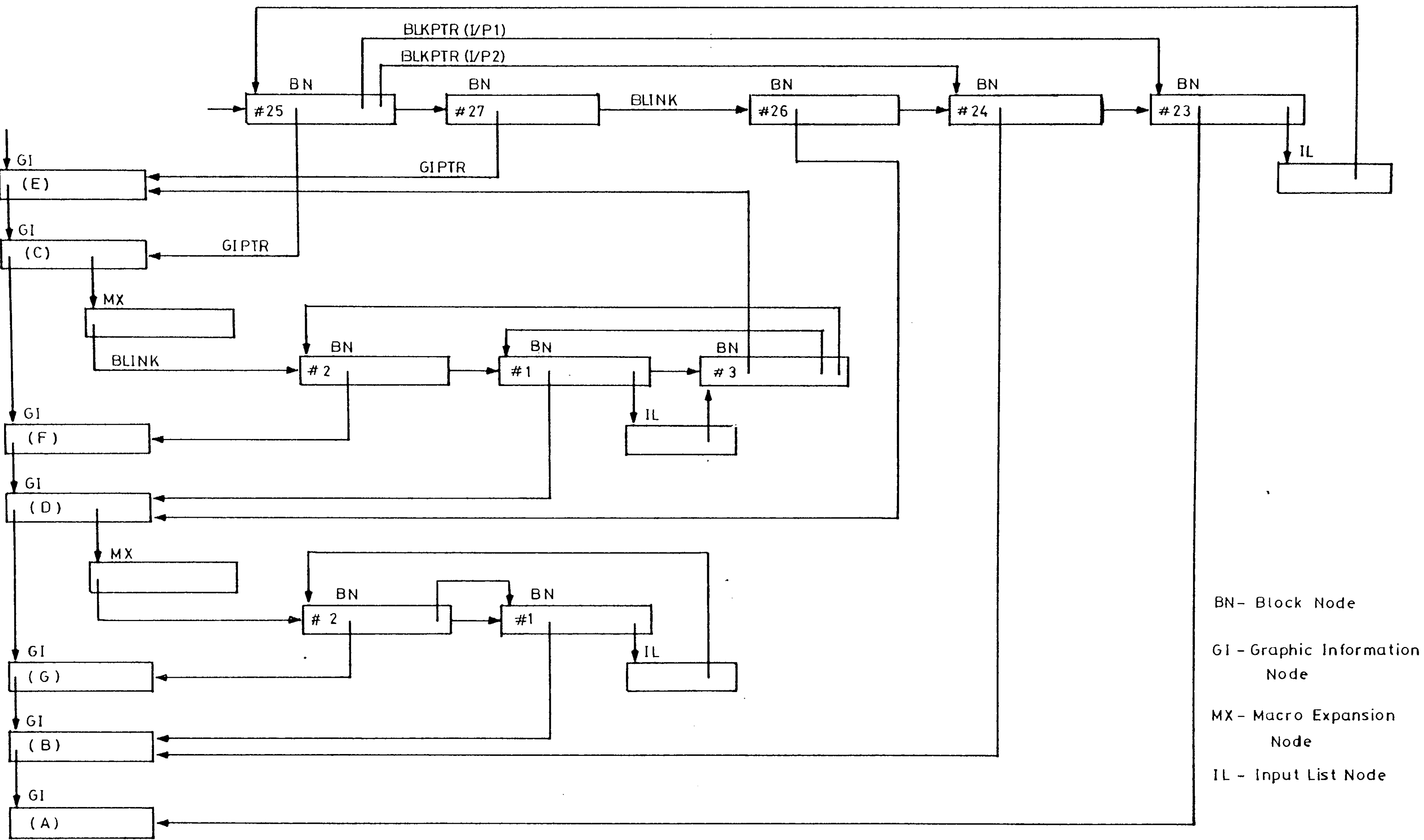


Figure 3.9 Components of Graphic Data Structure



(refer Figure 3.7)

Figure 3.10 Graphic Data Structure

block to which the particular terminal is connected. For each output, ILPTR points to a linked list of Input List nodes which links all the input terminals which it sources.

In the Graphic Information node, GILINK links all other Graphic Information nodes. To cater for composite blocks, two new items have been added. These are the CLASS and MXPTR values. The CLASS of a simple block is zero; that of a subpicture is 1 and that of a macro is 2. For composite blocks (both macros and subpictures) MXPTR points to the Macro Expansion node which relates a composite block's terminals to those of its constituent blocks.

TLINK of each Text node links all Text nodes belonging to the same block. More details on the Text node are given in Section 3.4.6 .

Each Input List node specifies an input terminal to which a particular output terminal is connected. All Input List nodes that specify inputs connected to the same output are linked by ILLINK.

3.3.5 Limitations of Macro Blocks

The Graphical Data Structure described so far has a shortcoming, related to macro blocks. Once a macro block has been defined, it may be used in several different places. While the definition of a SUBPICTURE may be changed, because it is a one-off structure, that of a MACRO must remain the same for all uses, since all similar MACRO blocks share the same definition. The macro feature of textual programming languages also operate in this way. In these languages the macro expansions may be made variable by parameterisation. In GPCL, this is equivalent to requiring all the different variables to be 'brought out' to the terminals of the macro block (that is, made available to external components). Thus, in the example of the cascade controller block (Fig 3.11a), the values of

P1, I1, D1 and P2, I2, D2 (representing constants) are the same for every use of the block, whereas for the block in Fig 3.11b the P, I and D values have been made external.

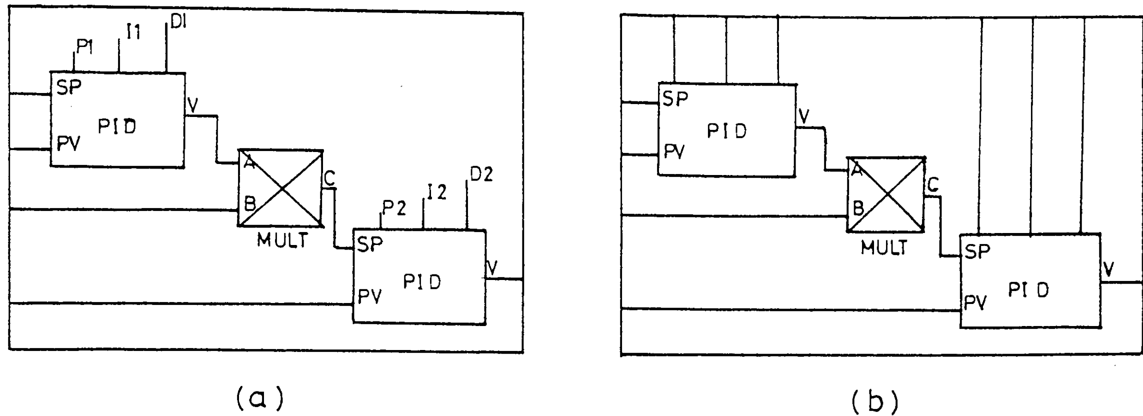


Figure 3.11 Cascade Controller Block

The hardware equivalent is fixed potentiometer settings in the first case, and linking the P, I and D signals together with the other signals through to an edge connector or backplane in the second.

Obviously the 'wiring' can get very complex if more blocks are involved, or if more than one level of macro is used.

Since with identical hardware modules it is possible to adjust potentiometers individually, this facility should also be available in GPCL. That is, individual constants may be changed, so that they may be different among similar macro blocks. This requirement has drastic effects on the Graphic Data Structure.

In the same way that the internal wiring of hardware modules may not be altered, so the internal connection configuration of the GPCL macro block must remain unalterable.

The other requirement peculiar to process control applications is the presence of engineering units or dimensions, for some (but not necessarily all) variables. Thus the output of one block may be in litre/min, that of another in lbs/in^2. For a system without macros this

poses no problems, as it is a simple matter to attach a text string to a block's output terminal. The constraints imposed by the Graphic Data Structure on constants within macros also applies to text.

The seriousness of this is more debatable, however. It may be argued that only the inputs from the process, and the outputs of controller, are of interest to the engineer or operator. The problem may be solved by forbidding input or output interface blocks to be embedded within macros. Again, the operator may be interested not so much in the absolute values of these variables as in percentages. Nevertheless, there will be occasions when the engineer will wish to monitor the value of an inner block, as in the case of computed (as opposed to measured) variables.

To cater for the presence of 'individual' constants and text inside macros requires a drastic modification to the Graphic Data Structure resulting in a significant increase in complexity and size, if these constants and text are to be correctly displayed by the Graphic Editor.

The Run-time Data Structure (Section 5.4) does not suffer from these limitations because all macros have to be expanded, and all blocks (within macros) are replicated. By resorting to a small amount of textual communication in the form of constant and text lists during the graphic compilation phase, it is possible to set these constants and text individually without any modifications to the Graphic Data Structure. The consequence of such a strategy is that constants within macros displayed by the graphic editor need not indicate the run-time values, and they may therefore be omitted.

3.4 Graphic Editor

The Graphic Editor enables the user to interactively create a process control diagram and subsequently modify it. Since the facilities it provides is dependent upon the hardware employed, a brief hardware description is given in the following section.

The term 'picture' will be used frequently in the remainder of this chapter. A PICTURE may be defined as a collection of related graphical elements which must be displayed together. All the lines and text which comprise the description of a single block displayed on its own form a picture. If a diagram shows several blocks connected together, then the diagram is the picture.

In the rest of this chapter, the simpler term 'editor' will be taken to imply the Graphic Editor.

3.4.1 Hardware Description

As mentioned in Section 3.1.2, storage displays have the disadvantage of not permitting dynamic movement of the image. However, they generally provide finer resolution, and for the resolution and information content offered are less expensive than refresh types [94].

If a storage display is used, blocks may not be 'dragged' about, nor may user prompts be implemented easily. Deleted blocks and connections do not disappear from the screen until the picture is redrawn. Because of the slow drawing speed of storage displays it is not practical to redraw the picture every time a deletion is made if the picture contains a lot of lines, so redrawing must be specifically requested by the user.

These are inconveniences which may be tolerable if the user interaction is properly designed.

An advantage of using storage displays is the ease with which the Graphic Editor may be adapted to handle refresh displays. For example, if the Graphic Editor is designed to work with a storage display, and a refresh display capable of emulating the former is connected instead, the editing actions will remain basically unchanged, even though a joystick might be replaced by a light-pen. Furthermore, by setting a software switch, the Graphic Editor may be made to redraw the picture after every alteration.

The graphic terminal used in this project was a Tektronix 4051. This is a Tektronix 4010-type device which is a widely used storage terminal, and which has been emulated on refresh-type terminals. The screen is divided into 1024 addressable points horizontally and 780 points vertically. The terminal operates in two basic modes -- an alpha mode and a graphic mode. In the alpha mode it works like an ordinary alphanumeric terminal except that characters may be positioned anywhere on the screen. Characters are generated by hardware in a dot-matrix format. In the graphic mode, lines are drawn by manipulation of an electron beam. A line is drawn from the current beam position when the terminal receives a string of four characters which specify the coordinates of the endpoint of the line. In this mode a graphic cursor is produced which can be manipulated by means of a joystick control; hitting any key then causes the terminal to send the coordinates of the cursor, as well as the character typed, to the computer. The graphic cursor is invoked by a command from the computer; unfortunately, it cannot be manipulated by the computer.

The joystick is not as convenient as a light pen, but the only consequence is slightly slower operator response.

3.4.2 Menus

For ease of use, the Graphic Editor provides a COMMAND MENU which lists the various functions provided. Functions provided by the Command Menu include saving or retrieving a diagram, displaying or editing a picture, creating a new block type or modifying an existing one, and modification of the GRAPHIC MENU described below.

When invoked, the options are displayed and the user is required to enter the number of the function wanted. An alternative method is to let the user point the graphic cursor at the required function. However, this is computationally more involved as it requires comparison of the cursor position with the known positions of the functions listed. Furthermore this method would be more convenient for the user only if a refresh terminal with a light-pen were used, as manipulation of a graphic cursor otherwise would not be as fast.

The GRAPHIC MENU is a display of all the different block types defined in the system. When the user wishes to create a block in a picture, he can specify the block type through the Graphic Menu. The TYPE number is displayed next to each block as additional information, and the base of each block is also marked.

The use of a Graphic Menu in this system presents some problems non-existent in normal literal menus. The blocks have various shapes and sizes; 'normal' function blocks may not be too dissimilar in this respect, but such assumptions cannot be made for pure graphic blocks (plant symbols etc.).

The blocks should be displayed in their normal size. A reduced scale would allow the menu to be displayed more compactly but this makes accurate pin-pointing difficult, and distorts the relationship between the block and its intended position in the picture. Similarly, all blocks should be displayed with the same scale (i.e. life-size) to avoid confusion.

This means that is is not normally possible to display all blocks within the area of one screen, and the menu has to be split into several pages. The Graphic Menu must be user-modifiable, for a user may introduce new block types; conversely he might not need certain types and may wish to discard them. He may also wish to move the more frequently used ones to the same menu page.

It is therefore necessary to extend the Graphic Data Structure to handle the Graphic Menu. The extra data required are the menu page number, position and block type. GIPTR points to the Graphic Information node previously described. The menu substructure (Fig 3.12) consists of Menu Block nodes and Menu Page nodes. MBLINK links together all Menu

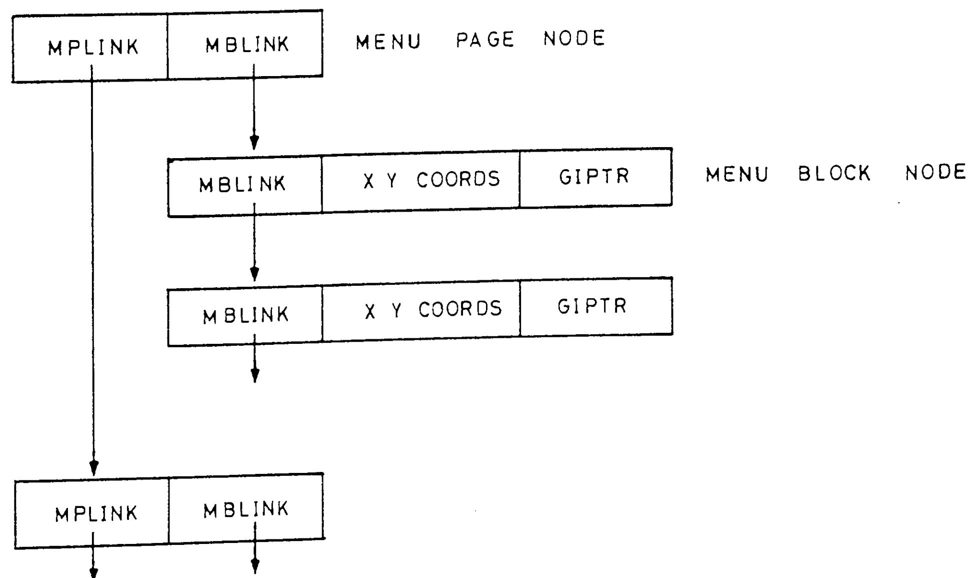


Figure 3.12 Menu Substructure

Block nodes belonging to the same page and MPLINK links all the pages of the menu. The page number is determined by the order of the Menu Page nodes.

Modification of the menu is achieved interactively with the user specifying the page number and position for each block type.

3.4.3 Editing Aids

Several facilities are provided by the graphic editor to facilitate the editing task.

To assist in the layout process, the user may cause a line grid to be drawn. There are systems where symbols are constrained to lie on discretely defined positions shown by a dot grid on the screen [99]. The advantage of such a system is to enable a very big user coordinate space to exist (possible screen positions may be described with fewer memory bits, thereby allowing a computer word to define a coordinate space which may be hundreds of times larger than the screen size), but such a picture size is not required in GPCL. The line grid aids alignment of points or blocks but does not place any constraints on the exact location of these objects.

Vertical and horizontal guidelines may also be drawn anywhere on the screen, for example through the output terminal of one block so that the input of another block to be connected to it may be aligned either vertically or horizontally.

These alignment aids do not affect the data structure whatsoever -- they are temporary objects which disappear when the picture is redrawn.

It is also possible to specify a picture which may be superimposed on the one being edited, in order to make use of some spatial

relationship that might exist between the two. This is one situation where the use of colours and different line intensities would be useful as it provides a clear distinction between the pictures. The superimposed picture may not be modified as its sole function is to assist in the layout of the picture being edited.

In contrast to the superimposition facility, a picture may also be used as a starting point for the creation of another -- in this case the data structure is affected as it involves creating a duplicate description of the original picture (called 'definition copying' in Sutherland's SKETCHPAD [80]).

3.4.4 Editing of Blocks

Editing a picture consists of the manipulation of blocks, connections and text. This and the following sections describe editing actions and assume that the editing mode has already been entered through the Command Menu.

In all cases the term 'pointing' refers to the action of positioning the graphic cursor and implies pressing a function key or special character if this is not explicitly stated.

To create a block in the picture, the user has to specify the type of block required. If such a type does not exist, it must be defined first (Section 3.4.7). If the type of block exists in the picture being edited, the user simply points (the graphic cursor) to an existing block of that type and specifies this as the type of the block he wishes to create. If no such block is present in the picture, he may call up a page from the Graphic Menu, and point to the base of the desired block. If however the user is familiar with the type number he may wish to specify the number directly, thereby avoiding the delay required to draw the menu

page. This is also permitted, and on entering the type number the relevant block is displayed.

The next step is necessitated by a shortcoming of storage type graphic terminals -- the inability to continuously manoeuvre objects on the screen. The user has to relate a certain point on the selected block to another point in the picture, in order that the block may be created in the correct position in one attempt. This is done by picking a reference point within or around the selected block (often an input or output terminal) using the graphic cursor, then pointing to the target position in the picture, where the block will subsequently be drawn.

When a block is created, it is automatically assigned a block number , which will be displayed. This number can be changed by pointing to it and entering a special character. The graphic editor will then allow a new block number to be entered.

JUNCTION blocks are a special case. They may be created in the correct position in a single step by positioning the graphic cursor and pressing a special key.

Moving a block in a picture involves a similar process. The user first points to the base of the block, then points to a reference spot around the block, and finally points to the new position for this particular reference. The necessity to point to the base first is to safeguard against moving the wrong block.

Deleting a block is much simpler. The user just points to the base of the block and presses the delete function key. The editor removes the Block node from the Graphic Data Structure, and deletes all connections that have been made to that particular block. However, on the screen the lack of selective erasure means that the picture is not redrawn after every modification since too much time would be wasted.

3.4.5 Editing of Connections

To create a connection between two terminals, the user first points to the output terminal, then to the input terminal, thereby specifying the direction of data flow as well as the terminals involved. It is possible for the graphic editor to work out the data flow direction by itself, but this obviously involves extra work especially if a chain of junction blocks is involved (i.e. a multi-segmented line).

The editor checks the connection for validity before updating the relevant input and output pointers in the Graphic Data Structure. For example, two outputs may not be connected together. It is also illegal to connect anything to an input if that input has a constant value attached. Checking has to be performed at this stage because the connection description explicitly defines the direction and nature of each connection. Terminals of incompatible signal type may also not be connected.

When a connection is made between the output of one 'normal' block (i.e. other than a JUNCTION) and the input of another, checking is trivial. This is not the case, however, when one or both of the blocks involved is a junction block, since its graphical representation is a big 'dot' with its input and output coincident. In this case, whether the connection being made is to the junction's input or output can only be determined in the context of the other connections to the junction, if any, and the nature of the terminal at the other end of the connection being made.

Another complication arising from junctions appears in the case of a multi-segmented connection line originally connected at one or both ends to normal blocks, which subsequently has the end connections deleted resulting in a multi-segmented line passing through several junctions.

The previous existence of a connection at one end of this line to a normal block has created a definite direction for the data flow. If a new connection is later made to a normal block, the data flow along the whole of the multi-segmented line will have to be reversed if the new terminal and the previous (disconnected) terminal are not of the same type.

Deletion of a connection is achieved by pointing to one terminal and pressing the delete connection key. It is important that the cursor be positioned reasonably accurately in order to avoid deleting an adjacent connection instead.

A delete connection action at a junction involves more processing and requires more care. The cursor must be positioned as near as possible along the line to be deleted, and at the same time a short distance away from the centre of the junction. The editor has to work out both the junction involved and the line to be deleted.

3.4.6 Editing Text

Text editing is a more complicated process than the editing of blocks or connections for the following reasons :-

(i) text (which as mentioned in Section 3.3.2 is always attached to blocks) may be divided into two types -- one appears on all blocks of the same type (type-specific) while the other is unique to each individual block (block-specific);

(ii) type-specific text includes the block type name and terminal names (if any) as well as perhaps some 'random' text ;

(iii) block-specific text may also be sub-divided into 4 types -- constants, engineering units, block name and 'random'

text;

(iv) 'random' text, which is for purely commentary purposes, may appear anywhere in or around the block (in contrast to the other types of text which have definite positions);

(v) constants have connection implications; an input cannot have a constant and simultaneously be connected to another block.

Type-specific text is defined when creating a new block type. When editing a normal diagram, type-specific text cannot be edited.

The block number may be edited by pointing to its first character and entering a special character. The alpha cursor will appear and the new number may be typed in. The editor has to check the validity of the text string and ensure that another block in the picture does not possess the same number.

The block name is edited by pointing to the base of the block and entering a special character. The editor will then allow the name to be entered next to the block number. If a name exists it will be overwritten. The user may also delete a name.

An output may be given engineering units by pointing to the terminal and entering a special character. The editor then allows the units to be entered next to the output, or next to the name of the output if the name exists. Again, units may also be deleted.

Type-specific text may only be edited during creation or editing of the block type (Sections 3.4.7, 3.4.8). To edit a block type name or a terminal name, the user points to the base of the block or the relevant terminal and enters a special character. The alpha cursor will then appear whereupon the name may be entered. The editing procedure is similar to that for block names and engineering units.

To edit random text, the user first specifies the block by pointing to its base. The computer then allows the pieces of text belonging to this block to be edited. If text exists, an alpha cursor will appear at the first character of the first line. Modification of text is achieved by simply overtyping. To proceed to the next line, a carriage return must be entered, and the alpha cursor will now appear at the start of the next line. This process is repeated until the last line, after which a graphic cursor will be produced.

A new line of text may be created by pointing to the desired position following which the text may be entered. After the line is terminated with a carriage return, the graphic cursor will re-appear and the process may be repeated.

If when the graphic cursor is present, only a carriage return is entered, it will be interpreted as the end of the text edit.

When the alpha cursor is at the first character of an existing line of text, it is possible to move or delete the line by typing a special character followed by a carriage return. This is interpreted as a 'move/delete string' command, and a graphic cursor will appear. Entry of a 'delete' character will cause the string to be deleted from the data structure. To relocate the string, the graphic cursor is moved to the new position and another special character entered. The line of text will be printed at the new position and the alpha cursor will re-appear in the first character position.

Constants are created by first pointing to the relevant input and entering a special character. An alpha cursor will appear at a suitable location near the input for the value to be entered. The editor has to check the numeric validity of the string entered and check that the input is not connected to some other block, then flags the relevant input descriptor in the Block node. The text string is stored in a Text node

which is flagged to indicate that it is a constant text, and to indicate the input to which it belongs.

Constants may not be moved. To modify a constant it has to be deleted first by pointing to the input and pressing the 'delete constant' key.

The Text nodes for the various types of text are shown in Fig 3.13 . TLINK is a pointer to the next related Text node. The length of the node depends on the length of the text string. If the node stores 'random' text, the x and y coordinates of the text (relative to the base of the block) will be stored in the 3rd and 4th words. By assuming a

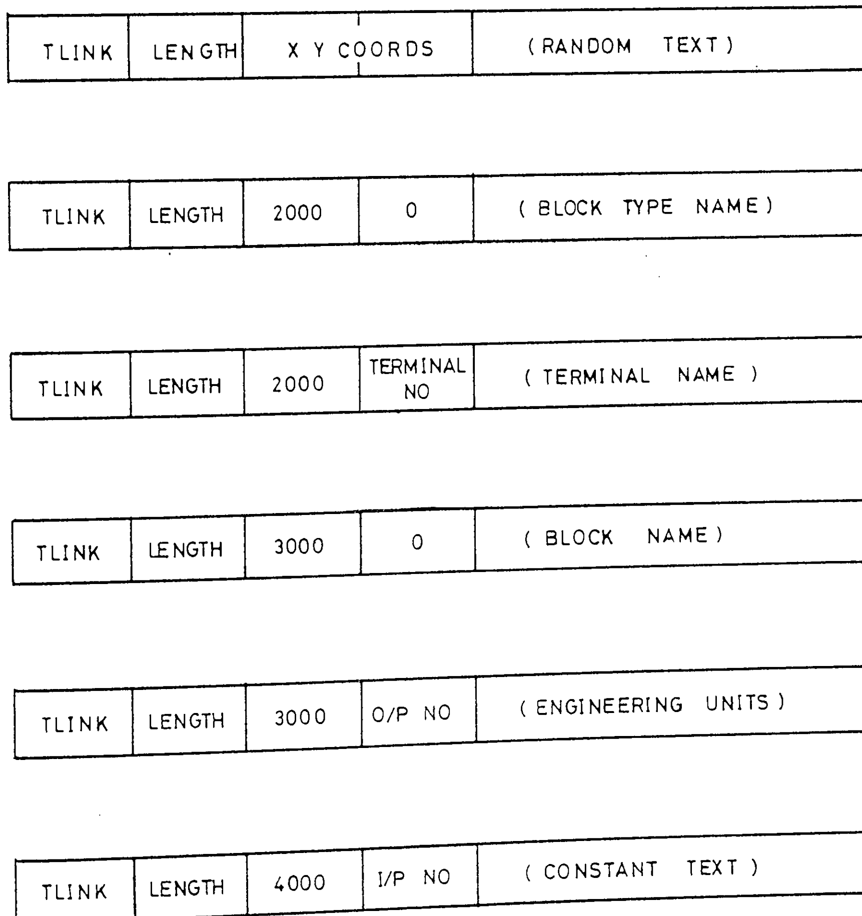


Figure 3.13 Text Nodes

practical limit on the coordinate values, numbers greater than this limit may be used to indicate other types of text. For example, function and terminal names are identified by a value of 2000 in the 3rd word, and the 4th word holds the terminal number which if zero identifies the text as a function name (block type name).

3.4.7 Editing Simple Block Types

Creation of a new simple block type means creating a new Graphic Information node, including possibly some text. Although this information may optionally be supplied directly as data by the user, it is normally achieved interactively using the Graphic Editor.

Creation of a new block type is a more complicated process than the previous editing functions. The first step is to define the shape of the new block. This is the only time when the block may be displayed larger than lifesize, to help produce a more accurate definition.

First the user has to define a base point -- the base of the block is the location to which all other coordinates are referred and also the position of the block type name (Section 3.3.3), if any. The user then defines the shape and all graphical details by drawing lines. The line drawing process is a simple one : the user points to the start position of a line, enters a 'start-of-line' character, then points to the end position of the line and types an 'end-of-line' character. Deleting lines involves a similar procedure.

It should be noted that at this level the only graphical entity existing is the LINE -- no 'subpicture' (as defined in SKETCHPAD [80]) facility is provided. The editor maintains a simple temporary data structure consisting of a linked list of pairs of endpoints (Fig 3.14) for this line-drawing exercise.

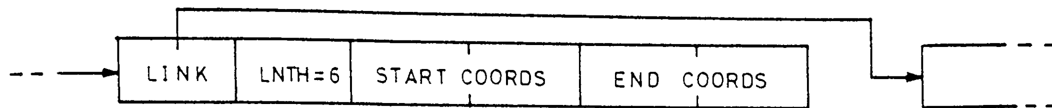


Figure 3.14 Temporary Line Drawing Data Structure

Simply creating all the lines is not sufficient, however, because the picture coordinates in the Graphic Information node not only define the lines, but also the order in which they are to be drawn (Section 3.3.3). After having created all the lines, the user has to specify the order of 'draws' and 'moves'. This is facilitated by the graphic editor displaying the block in one half of the screen : as the user specifies each endpoint in turn (also indicating if it is a draw or a move), the actions are echoed in the other half of the screen.

Having created the shape of the block, the user has to define the position of the input and output terminals, if it is a function block. This is done by pointing to the required spots and pressing the 'create input' or 'create output' function key. The sequence in which they are created determines their terminal numbers starting from 1 for both inputs and outputs. The number will appear in the picture next to the spot where the terminal is defined (marked by a cross), for the benefit of the user.

Terminals may not be deleted individually. If too many terminals have been created, specifying the 'delete terminals' action will remove all terminals. This is the simplest way to ensure that terminal numbers are always consecutive.

Terminals may also be re-positioned. This may be used to re-order the terminals if they have been created in the wrong order.

Text (in this case always type-specific) may be created and edited in a similar way to that described in Section 3.4.6 . However, this can only be done when the block is drawn unmagnified. Similarly, the position

for the block number must be defined when the drawing is life-size.

At the end of the creation process, the editor will request a TYPE number for the block type. It then forms a new Graphic Information node and includes it into the Graphic Data Structure, along with any text that has been created. The user is also asked to supply the non-graphical data for the new block type so that a Non-Graphical Data node may be created for it.

Modifying an existing block type is a similar process. The picture coordinates are expanded into the temporary data structure if lines are to be edited. The only restriction is that the number of inputs and outputs cannot be changed if a block of that type is used anywhere.

3.4.8 Editing Composite Blocks

To create a composite block a picture must have previously been created showing all the component blocks. The user can indicate that he wishes to define the picture as a composite block. The editor then allows him to define the graphical information for the new block in exactly the same way as for creating a simple block type.

After this has been completed, the user has to relate the terminals of the newly-formed composite block to those of its constituent blocks. This is done by displaying the picture to be used. The input and output terminals corresponding to those of the composite block may then be specified in the correct order (similar to the procedure for defining the terminals for a simple block).

With this information the editor is able to create the Macro Expansion node for the new block, as well as the Graphic Information node, and establish the links between them and the blocks that make up the picture. The other piece of information that must be supplied is

whether the block is to be a subpicture or a macro.

Subpictures may be modified like a normal picture; however, the inputs and outputs corresponding to the terminals of the subpicture block may not be changed.

A macro block can only be modified if it is not being used in any picture.

Recursive definitions are not allowed. For example it is not permitted to define a block type 'A' which contains a composite block type 'B' which in turn contains a block type 'A'. Since block type numbers are unique, this is easily checked.

CHAPTER 4

FUNCTIONAL BLOCKS

4.1 Introduction

The basic set of blocks used simulate all the common hardware analog and digital modules familiar to process control engineers. Some frequently used blocks are shown in Fig 4.1 . The objective of this

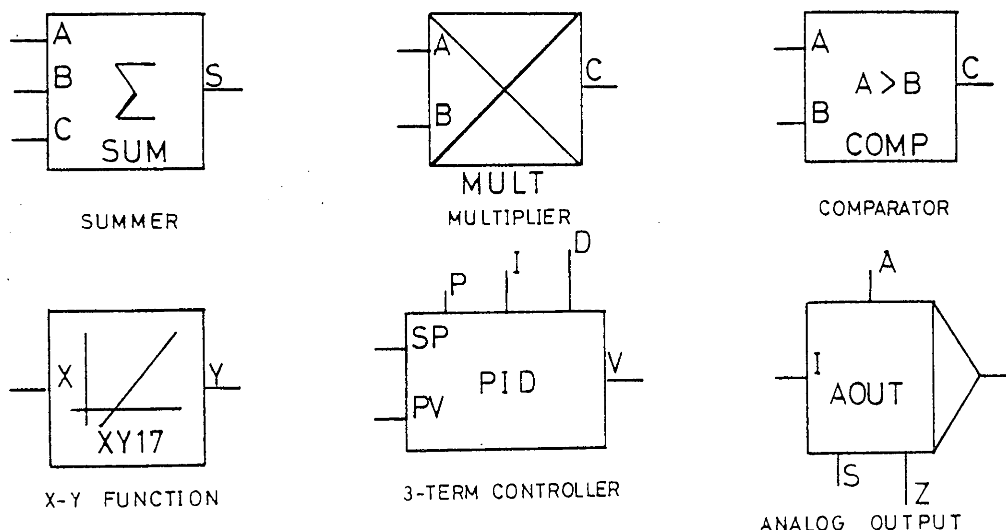


Figure 4.1 Examples of Functional Blocks

language, however, is not to attempt to provide a complete set of functional blocks to cover all the requirements of process control applications. While a comprehensive set of functions is convenient, a more important requirement (applicable to any language) is the facility for the user to define his own functions.

In GPCL, there are two ways by which a new functional block may be defined. The first involves creating a subpicture using already existing block types and defining the subpicture to be a composite block. This is

a purely graphical method and has already been described in Section 3.4.8 .

The second method is used to define a new simple block. Use cannot be made of other existing blocks since it is not a composite block.

The graphical characteristics of a block have already been discussed in Chapter 3, as has been the process of defining the graphical information for a new block type (Section 3.4.7). The following sections will deal with the functional properties of simple blocks, the format of the CORAL 66 procedure which defines the function of the block and the dependence on the storage structure for its variables, and the implementation of several block types.

4.2 Functional Characteristics

Every block type has a certain number of inputs and outputs, although it is not necessary for both to exist. The relationship between the inputs and outputs are defined by a CORAL 66 procedure, which may be as simple as a one-line arithmetic expression, or as complex as a set of state equations used in modern control theory.

In the run-time system, only one copy of the procedure for each block type is stored, this procedure being shared by all occurrences of the particular block type. Consequently all program variables whose values are to be retained until the next time a block is processed must be saved. Thus apart from the output values, some internal variables may have to be stored. The number of internal variables for each block type must be known in order that space may be allocated for them in the Run-time Data Structure (Section 5.4).

Before execution of the actual process control system can start, it is necessary to have determined the order of processing of the blocks,

which depends on the specific interconnection of the blocks. A block can only be processed when all its inputs are defined. Input interface blocks are therefore processed first, then subsequent blocks as their inputs become defined, and finally ending with the output interface blocks. However, blocks are sometimes connected into closed loops, and the processing sequence can only be determined with a knowledge of certain properties of the blocks. A block type may have some of its outputs initialised before execution begins. These outputs are defined in a RESET OUTPUTS FLAG word (ROF). Each output of the block is associated with a bit in the ROF word; if the output is initialised, the corresponding bit is set to '1'.

Some function blocks use expressions which depend on past values of input to obtain new values of output. Such blocks are called DYNAMIC blocks; an example of a dynamic block is an INTEGRATOR using the trapezoidal integration algorithm

$$I_n = I_{n-1} + T(U_{n-1} + U_n)/2$$

where I_n is the nth output value and U_n, U_{n-1} the current and previous input values respectively.

It is possible to define dynamic blocks where in the evaluation of the current output, the values of input used are restricted to past ones only. An INTEGRATOR using the rectangular integration algorithm

$$I_n = I_{n-1} + TU_{n-1}$$

has this property. Blocks of this type will be referred to as RETROSPECTIVE; they have special significance in the determination of the execution sequence as will be shown in Section 5.3.1 .

Input interface and output interface blocks (Section 4.5) which interact with physical devices also affect the processing sequence. These and the retrospective property are indicated in a GENERAL FLAG word (GF) as shown in Fig 4.2 .

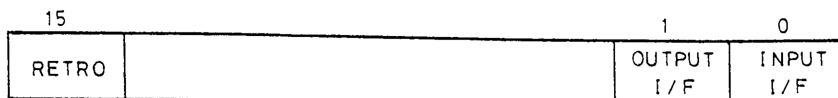


Figure 4.2 General Flag Word

Input and output signals may be classified as one of two types -- continuous or logical. Logical signals can take on only two values, 1 or 0 (or TRUE or FALSE). In order to prevent connections between terminals of different signal type, it is necessary to provide signal descriptions for both inputs and outputs. These take the form of two flag words, the LOGICAL INPUTS FLAG word (LIF) and the LOGICAL OUTPUTS FLAG word (LOF). Each input (output) terminal has a bit in the LIF (LOF) word which is set to '1' if the signal type is logical.

Inputs may be further grouped into two types -- 'parameters' and 'normal' inputs. The former only accept constant values and may not be connected to the outputs of other blocks; the latter do not have such a restriction on values. In the GPCL run-time system, inputs with constant settings are treated in the same way as all other inputs, and are connected to a CONSTANT POOL BLOCK which is a function block with no input but numerous preset outputs. This treatment allows any input to be connected to either a constant value or the output of any block. In order to prevent the connection of a 'parameter' input to another block, the CONSTANT INPUTS FLAG word (CIF) is used to define all those inputs which require constants.

The number of internal variables (NIV), the General Flag word (GF), the Reset Outputs Flag word (ROF), the Logical Inputs and Logical Outputs Flag words (LIF, LOF), and the Constant Inputs Flag word (CIF) form the Non-Graphical Data (NGD) alluded to in Section 3.3.3 . Each block type has a record in the NGD Table which is organised as a linked list of NGD nodes (Fig 4.3). NGDLINK points to the next NGD node.

NGDLINK	LENGTH =8	NIV	GF	ROF	LIF	LOF	CIF

Figure 4.3 Non-Graphical Data Node

4.3 Procedures

The functional relationship between the inputs and outputs of a function block is defined by an algorithm written as a CORAL 66 procedure. The method of accessing variables is dependent upon the organisation of the Run-time Data Structure (Section 5.4), and a discussion of this will be deferred until Section 5.4.4 . For the purposes of describing the procedure in general, it suffices here to refer to each variable by meaningful names (e.g. INPUT2, OUTPUT etc).

Two examples of procedures are shown in Fig 4.4 . The identifiers INPUT1, INPUT2, OUTPUT, RESET, and TIMECONSTANT are all CORAL 66 macro names. They allow convenient names to stand in place of the actual identifiers. The 'DEFINE' statements are the CORAL 66 macro definitions, the details of which are not shown (see Section 5.4.4). The 'DELETE' statements cancel these definitions.

It should be noted that the procedure headings do not include a parameter list. All variables a procedure needs to access are global. An example is the variable called INTERVAL which depends on the time interval between sucessive executions of the algorithm.

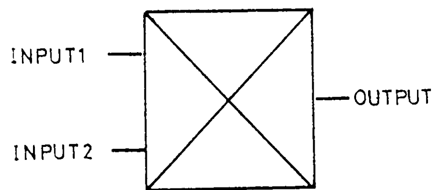
Recursive procedures should not be used as they involve varying memory allocation and are therefore a potential source of danger in a system with limited memory.

```
'PROCEDURE' MULTIPLIER;

'DEFINE' OUTPUT "- - - -";
'DEFINE' INPUT1 "- - - -";
'DEFINE' INPUT2 "- - - -";

'BEGIN'
        OUTPUT:= INPUT1*INPUT2
'END';

'DELETE' INPUT2; 'DELETE' INPUT1; 'DELETE' OUTPUT;
```




```
'PROCEDURE' INTEGRATOR;

'DEFINE' INPUT1 "- - - -";
'DEFINE' TIMECONSTANT "- - - -";
'DEFINE' RESET "- - - -";
'DEFINE' OUTPUT "- - - -";

'BEGIN'
        'IF' RESET > 0 'THEN' OUTPUT:=0

            'ELSE' OUTPUT:= INPUT1*INTERVAL/TIMECONSTANT
                            + OUTPUT

'END';

'DELETE' OUTPUT; 'DELETE' RESET; 'DELETE' TIMECONSTANT;
'DELETE' INPUT1;
```

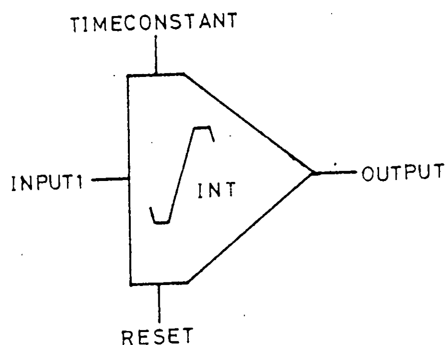


Figure 4.4 Examples of Function Block Procedures

4.4 Internal Representation of Variables

A question that arises with regard to continuous variables is whether they should be represented by fixed or floating point numbers. The three factors to be considered are :-

(i) storage requirements

(ii) execution speed

(iii) dynamic range .

Another question that arises is how logical variables should be handled.

The following discussion is based upon the assumption of a 16-bit processor being used in the process control computer. Although there exist some process computers which employ 8-bit microprocessors, the majority (with the notable exception of the 12-bit PDP8 series) use 16-bit word lengths. This is increasingly true even for microprocessor-based systems due to the number of 16-bit microprocessors becoming available.

4.4.1 Storage Requirements

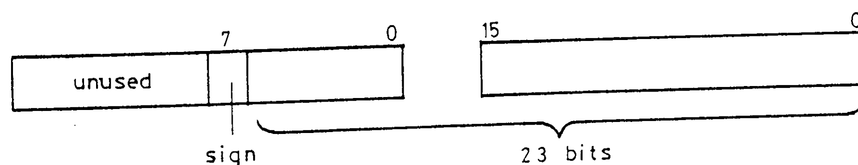
For a given resolution, fixed point numbers require fewer bits than floating point ones, but in practice this difference can not necessarily be used to advantage.

The resolution of the common 12-bit analog-to-digital and digital-to-analog converters is 0.025% of full scale. To reduce computational errors due to either overflow or truncation, variables have to be stored with a few more bits. If a total of 15 bits is sufficient (with the 16th bit being a sign bit) then for a 16-bit processor the fixed point number can be stored in one (16-bit) word, whereas a floating point number would need part of an extra word to store the exponent if the same resolution is to be preserved.

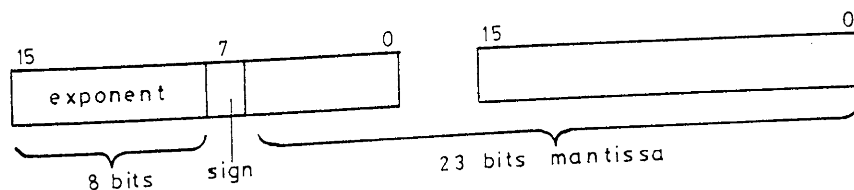
If however, 15-bit arithmetic is deemed to give either insufficient resolution or dynamic range, then the fixed point number must be held in more than one word. Normally the number of additional bits required will be rather less than a complete word. For example, 23 bits will give a resolution of 1 part in 8 million; if this is adopted, the remaining 8 bits of the additional word will be left unused (see Fig 4.5a). Use of the entire width of the two words does not provide any useful increase in accuracy.

The same number may be stored in floating point format in the same space, with the remaining 8 bits holding the exponent (Fig 4.5b).

Thus, unless variables are scaled in fixed point representation to occupy one word only there is no storage penalty in using a (two word) floating point format. In this implementation it is assumed that floating point representation will be used.



(a) Integer/Fixed-Point



(b) Floating Point

Figure 4.5 Examples of Number Representations

4.4.2 Speed

The instruction set of most 16-bit processors include instructions for integer multiply and divide. Single precision fixed point arithmetic requires little more than integer arithmetic plus some shifts, and is therefore fairly fast.

Double-precision fixed point arithmetic takes longer and relies on the presence of double-precision integer instructions, the absence of which will necessitate more code and increase execution time significantly.

Floating point arithmetic poses more problems -- many 16-bit processors do not have floating point instructions. For those which do not have them relatively lengthy software routines must be used. Alternatively, separate floating point hardware may be incorporated. Floating point instructions, if they exist, are slightly slower than integer ones, but the difference in speed becomes less significant when all the other operations the processor has to perform is taken into account.

4.4.3 Range

The range of values that can be represented by a particular fixed point format is naturally limited, whereas floating point numbers have a range which is more than sufficient for most practical purposes. Use of the latter therefore imposes virtually no restrictions, whereas range is an important consideration when fixed point numbers are used.

However, the problem of range limits also exists in all analog equipment, including process control hardware, and it has always had to be taken into account in the design process. Furthermore, the whole function of the process controller is to produce an analog output varying

within a defined range, e.g. 0-10V , 4-20mA. Similarly, analog transducer signals are normally scaled to range over a limited span.

Nevertheless, there is no intrinsic requirement to carry over the notion of such span limits to the blocks internal to the controller diagram although this could be achieved by limiting the output range of individual blocks. The preferred alternative is to use floating point representation for all variables, with sufficient range for scaling between blocks to be unnecessary. Scaling is then only required in analog input/output interface blocks.

Also, the theoretical speed advantage of fixed point arithmetic made possible by the application of scaling is offset by the very act of scaling itself.

4.4.4 Logical Variables

Not all functional blocks process continuous variables; some blocks are required to execute logical functions. An example is the COMPARATOR block which compares two continuous signals at its inputs and produces a logical output (1 if one input is greater than the other, 0 otherwise).

In processing such Boolean variables, maximum storage efficiency is achieved by using only one bit of storage to represent each variable, thereby storing as many variables in each computer word as there are bits in that word. However, efficiency of storage is not related to efficiency of execution, and referencing these variables requires more code than if each variable occupied a separate word.

CORAL 66 itself does not provide for a BOOLEAN data type. In this case, as in some other languages, it is simplest to use an integer to represent a single variable, using perhaps zero for the '0' state and a

non-zero integer for the '1' state. The disadvantage of inefficient memory usage on the one hand is offset on the other by the more straightforward referencing.

To preserve the homogeneity of the Run-time Values Table where all a block's variables are stored (Section 5.4.1), logical variables are not afforded special storage. No distinction is made between logical and continuous variables in the Table. The '0' state is represented by a floating point value of zero and the '1' state by any other value, normally one. The seriousness of storage inefficiency depends on the actual representation of floating point number which is implementation-dependent, as well as the number of logical variables present.

Although both continuous and logical variables are represented by floating point numbers, this does not prevent type checking on signals to be performed to prevent incompatibility, resulting for example from attempts to connect an analog signal to a logical input. Type checking is enabled by the presence of signal description flag words in the Non-Graphical Data as described in Section 4.2 .

4.5 Some Implementation Considerations

Function blocks may be divided into 2 categories -- computational blocks and input/output interface blocks. Input/output interface blocks are those that are connected to the physical process interfaces (e.g. analog inputs and outputs), and are machine dependent to a certain degree. Computational blocks are ones that do not interact with the physical interfaces, and are therefore machine-independent. I/O interface blocks may also involve a certain amount of computation such as scale and offset adjustments and sometimes linearisation.

The implementation of most computational blocks is quite straightforward and obvious (as in the examples of the MULTIPLIER and INTEGRATOR in Section 4.3). A few computational blocks however, present more than one possible method of implementation. These, and some typical I/O interface blocks, will be discussed in the following subsections.

4.5.1 Constant Pool Block

As stated in Section 4.2 constant inputs to blocks are implemented as inputs connected to one of the outputs of the Constant Pool Block. Since connections may be altered and blocks added or deleted in the run-time system, the number of constants will vary. This suggests either a single large, variable-sized Constant Pool Block (i.e. having a variable number of outputs), or a number of suitably sized fixed ones. The former approach eliminates wastage, but necessitates re-allocation of the Run-time Values Table entries (Section 5.4.1) whenever the number of constants change, which is undesirable. The second approach is preferred.

4.5.2 X-Y Function Block

The X-Y function block (Fig 4.6a) holds a set of x and y values which define a curve. A value of x present at the input will produce an interpolated value of y at the output. The values of x and y are stored in the Run-time Values Table as internal variables of the X-Y function block (Fig 4.6b).

To provide only one size of block (a fixed number of data points) is undesirable since it may not provide the accuracy required; conversely not so many points may be needed. Here again the possibility of variable-sized blocks (for storage of values) suggests itself, as a means of minimising storage requirements. This requires the number of points (or

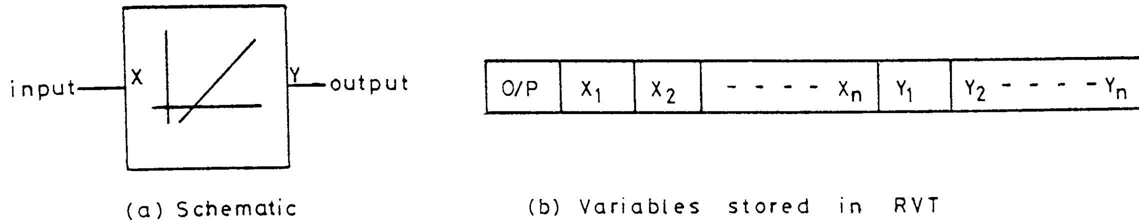


Figure 4.6 X-Y Function Block

the number of internal variables) to be stored somewhere other than in the Graphic Information node in the Graphic Data Structure, and other than in the TYPE node (Section 5.4.3) in the Run-time Data Structure, since these nodes hold information which is identical for all blocks of the same TYPE.

One possible solution is to provide the number of points as another input to the block (Fig 4.7). This input must be defined as a constant parameter input to prevent it being connected to some other block. Also, precautions have to be taken by the Supervisor in the run-time system to prevent this value being changed.

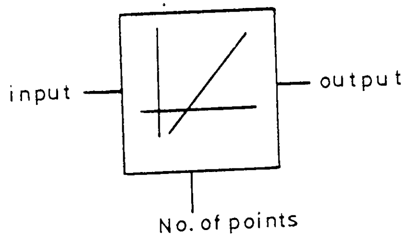


Figure 4.7 Variable Size X-Y Function Block

The use of variable length blocks complicates the Run-time Data Structure as well as the Supervisor functions, since it cannot now simply rely on the values of the numbers of outputs and internal variables in the TYPE node to determine the actual number of variables possessed by the function block.

By providing several X-Y function block types differing only in the number of data points, it is possible to avoid undue wastage without the complications of variable length blocks. Only one interpolation routine is necessary for the different sizes, as the routine can access the TYPE information in the Run-time Data Structure to determine the number of points.

The setting up of the x and y values may be done using preset lists which are supplied for inclusion during the compilation phase.

X-Y-Z function blocks are implemented in a similar way.

4.5.3 Delay Block

Delays may be implemented using either a continuous analytical approximation of the Padé type [100,101] or an N-stage shift-register type delay block which provides a true transport delay. The former requires less storage, but is less accurate, whereas the latter provides a better simulation at the expense of more storage.

A method of implementing a N-stage delay using an M-stage shift-register is shown in Fig 4.8 . The input variable enters the M-stage shift-register at a point that is N-1 stages behind the output, and gets

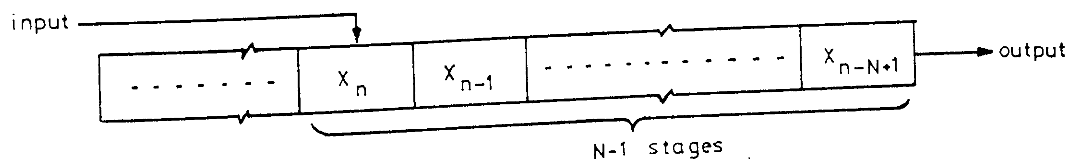


Figure 4.8 Variable Stage Delay

shifted along until it eventually emerges at the output. By parameterising the value of N, delays from 1 to M stages may be obtained (Fig 4.9a). Further, N may be made variable by connecting this input to another block, but precautions must be taken to ensure that the value

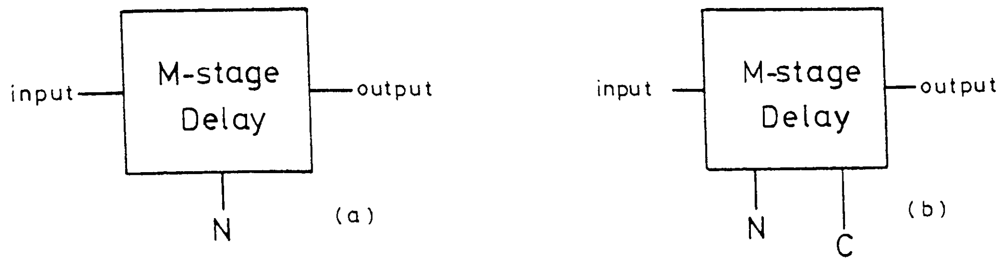


Figure 4.9 Delay Blocks

does not exceed its permissible range. Also, the value of N must be altered very gradually in order to avoid discontinuities.

Since the number of stages required depends on the frequency of processing, large delays may demand excessive memory storage. To cater for 'large' delays, therefore, another parameter C may be introduced which determines the number of processing intervals before the values are advanced by one stage (Fig 4.9b). This reduces storage requirements at the expense of some loss of resolution. Interpolation may be used to provide a smooth output change. As with N, the value of C may be made variable, with similar precautions.

By providing a few delay blocks with different numbers of stages, all delays may be catered for.

Fig 4.8 does not imply that all the values have to be physically shifted every C processing intervals. To minimize execution, the block is implemented as a circular buffer, with pointers to the first and Nth stages. The variables storage for the block is shown in Fig 4.10 . COUNT is incremented until it equals the input C, then the input and output pointers are incremented.

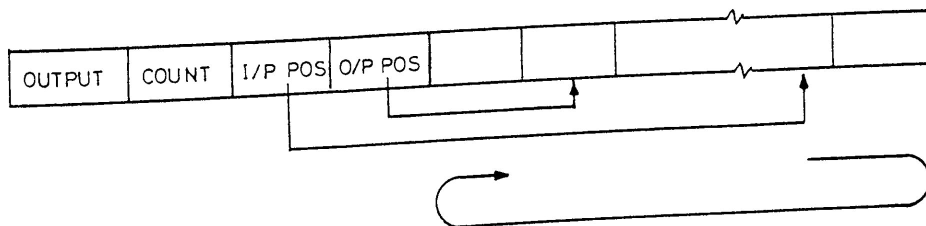


Figure 4.10 Storage of Delay Block Variables

As with the X-Y function block, it is possible to implement the delay block as a variable-sized block. However, in addition to the storage allocation and associated problems, the run-time size must be a constant. It is preferable therefore to provide only fixed size delay blocks.

4.5.4 Input/Output Interface Blocks

Since these blocks involve physical devices, the routines for these blocks are necessarily partially machine-dependent, although this dependence may be minimised through modularisation. Basically, two strategies may be used to process these blocks. The first is to use a separate input/output processing subsystem which scans the inputs at regular intervals and updates the relevant values in the Run-time Data Structure by Direct Memory Access. Output values may be transferred to the physical interfaces by the I/O processing subsystem in a similar way. This places a minimal processing demand on the main processor and is the fastest way of effecting process I/O. It is achieved at the expense of extra hardware and is only necessary when scan rates (and processing frequency) are high and a large amount of I/O is involved.

In the second method, input and output actions are treated as part of normal processing of the blocks. Some typical I/O interface blocks will be discussed.

4.5.5 Analog Input/Output Blocks

The Analog Input block is used to input a measured process variable. The actual hardware involved in obtaining the signals and the preprocessing of the signals before actually presenting them to the computer vary between different systems. Fig 4.11 shows a typical scheme.

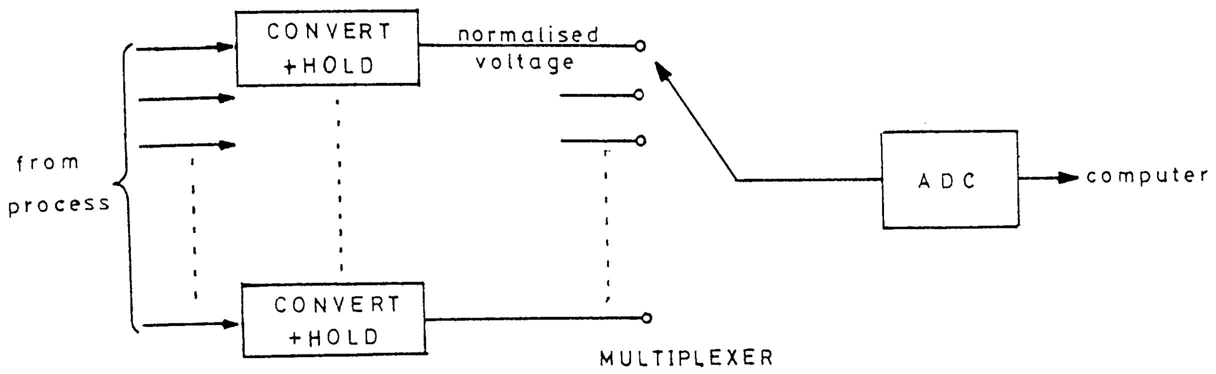


Figure 4.11 Typical Input Scheme

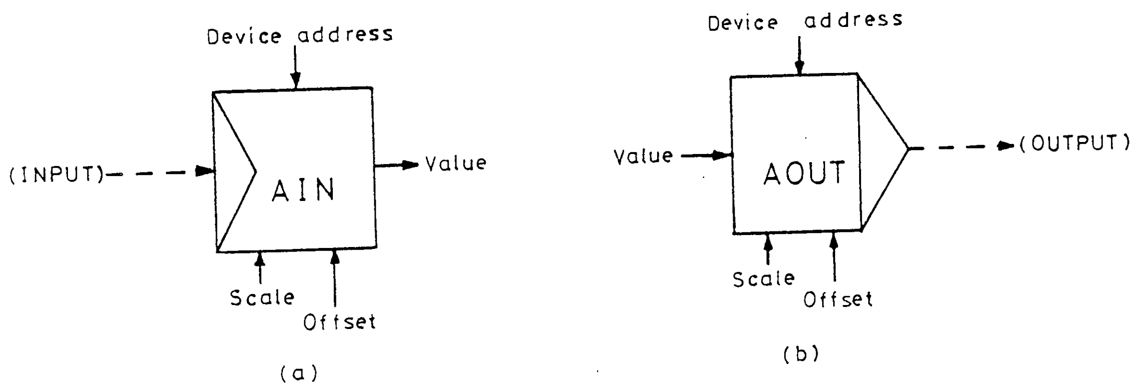


Figure 4.12 (a) Analog Input (b) Analog Output

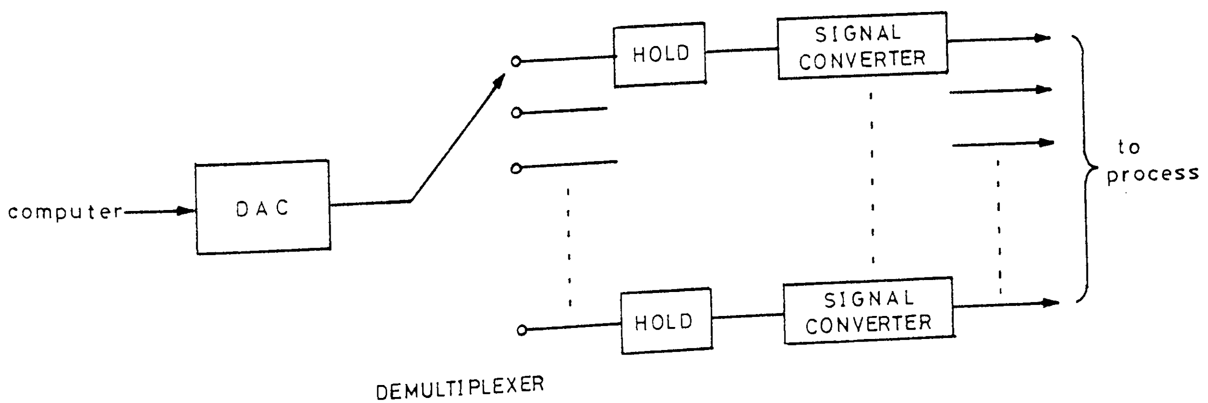


Figure 4.13 Typical Output Scheme

The signal from the transducers may be one of several types (e.g. 4-20mA, +10V to -10V, ac or dc, etc). Whatever the type, it may be converted into a normalised voltage which is then suitable for input to an analog-to-digital converter (ADC). Since the ADC is a fairly expensive device, the inputs would normally be multiplexed to share the same ADC [102].

In this example, the process computer has to switch the multiplexer to the wanted input, initiate the A-to-D conversion and wait for its completion (tens of microseconds). In such a case, the relevant input parameters to the Analog Input block (Fig 4.12a) are the address of the multiplexed input, and the SCALE and OFFSET values. The dotted line marked '(INPUT)' is a line that may be drawn to pictorially link the input block to a plant symbol. No real terminal exists or is required for this purpose.

The SCALE and OFFSET values convert the output of the ADC into a meaningful engineering value. In a special purpose analog input block extra processing may be involved, for signal conditioning. The choice of processing actions depends on the specific application.

Analog Output blocks (Fig 4.12b) function in a similar fashion (Fig 4.13). Computed output values may be scaled and offset before presenting to a digital-to-analog converter (DAC), the output of which is demultiplexed into several track-and-hold devices. The signals are then converted into the required signal type to drive the output devices. As with the Analog Input block, the dotted line in Fig 4.12b indicates a pictorial link to a plant symbol. The block does not possess a real output terminal in the data structures.

4.5.6 Digital Input/Output Blocks

As with previous blocks, these are machine dependent blocks.

Digital Input blocks are typically of the form illustrated in Fig 4.14a showing a single input line whose address must be provided, or that shown in Fig 4.14b where each block handles multiple lines (the latter being more suitable if the lines may only be addressed collectively). In the latter, each line involves one location in the Run-time Values Table.

Digital Output Blocks are the reverse of the digital input blocks (Figs 4.15a,b).

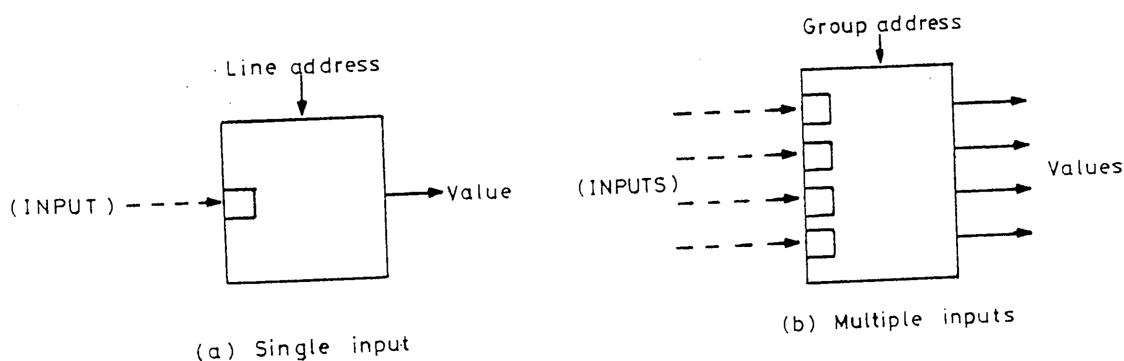


Figure 4.14 Digital Input Blocks

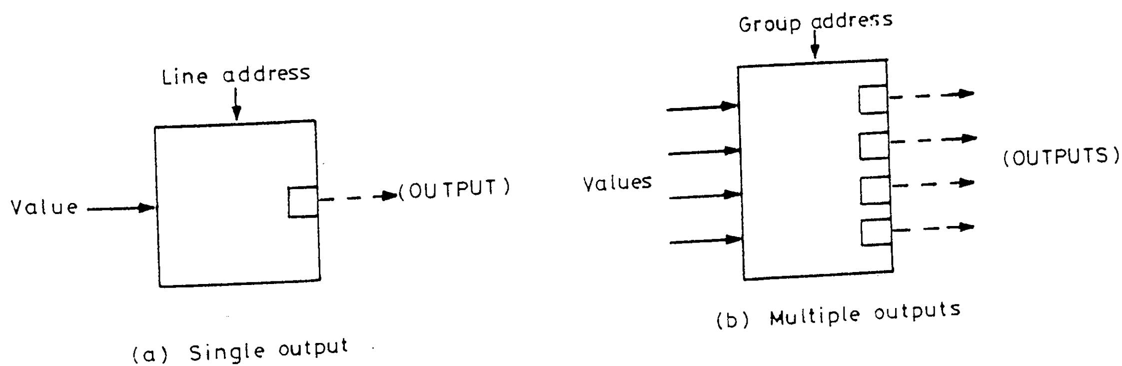


Figure 4.15 Digital Output Blocks

CHAPTER 5

COMPILATION AND RUN-TIME STRUCTURE

5.1 Introduction

The several stages involved in the process of arriving at the control program for the target process controller from creation of the block diagram have been enumerated in Section 2.3 . The first step, synthesis of the block diagram on a graphics terminal, has been dealt with in Chapter 3. The result is a data structure which can be input to the GPCL compiler, together with non-graphical information, to produce a control program in CORAL 66. Up to this stage the control program is still machine independent provided there are no assembly language inserts in any of the procedures.

Before describing the control program in more detail, it is appropriate to consider the objectives of the run-time system. The requirements of the run-time system are different from those of the graphics system. Since the run-time system is to be implemented on a fairly small machine, memory is more limited. At the same time, execution of the control program should be as fast as possible since it is performing a real-time task.

As stated in Section 2.1.7, modification of the control strategy in the run-time system is facilitated by making the execution data-driven. The facility to permit modification requires that sufficient information be retained in the run-time system.

Software and hardware resources needed to support a full graphics facility are normally beyond the capacity of the process control

computer, and this facility must therefore be forgone in preference of a much simpler one sufficient to allow on-line modifications. Thus, in the absence of graphics support, all purely graphical information (including part of text) will serve no active purpose in the run-time system, and therefore may be omitted.

To allow modification of software at such a low level has its dangers. The control strategy may be altered without causing a corresponding alteration in the source files. Interpretive languages are less seriously affected, since the new control scheme is apparent from the program statements, and it is merely necessary to obtain a new listing whenever changes are made, to update the source documents. Alterations made to normal non-data-driven compiled programs take the form of machine code patches, which are hard to detect and document if the change is not noted immediately. Consequently, modifications should be effected only at the source code level, which is then recompiled.

In GPCL, the control strategy is completely defined by data, from which the up-to-date version of the source may be recovered via a process called reverse compilation. The reverse compiler should not reside in the run-time system -- the task is performed by the more powerful graphic system running on a more adequately equipped computer.

5.2 Components of Control Program

The control program is not a single monolithic program; rather, it comprises a suite of program and data modules which can be divided into three components :-

(i) Supervisor

(ii) control algorithms

(iii) run-time data .

The SUPERVISOR is the program which is in overall control of the execution of the process controller. Its functions include :-

(i) determining the correct order of processing for the blocks;

(ii) calling the appropriate routine for each block and supplying it with the necessary parameters;

(iii) interacting with the operator, displaying values and enabling modifications;

(iv) performing checks to ensure proper operation of the system;

(v) providing logging and alarm facilities.

The CONTROL ALGORITHMS perform the actions required of each functional block, and have been discussed in Chapter 4.

The RUN-TIME DATA can be divided into four types :-

(i) data defining the characteristics of each functional block type;

(ii) data that provide interconnection information;

(iii) all the variables belonging to the functional blocks;

(iv) text associated with certain blocks.

The data are stored in the RUN-TIME DATA STRUCTURE (Section 5.4).

5.3 Compilation From Graphics

Two compilations are required in order to arrive at the final object code for the process controller. The first, performed by the GPCL compiler, takes data from the Graphic Data Structure and with additional non-graphical information, produces a control program in CORAL 66 code which includes the Run-time Data Structure. The second compilation is performed by a CORAL 66 compiler on the Supervisor and function block

algorithm programs to produce either assembly language or machine code for the target processor, and is beyond the scope of this thesis.

In the process of creating the Run-time Data Structure, the GPCL compiler has to expand macros and subpictures, sequence the blocks, initialise data in the various tables and allocate space for run-time variables. It also performs error checking and brings any errors to the attention of the user.

5.3.1 Sequencing

When the output of one block feeds into the input of another, the former must be processed before the latter. Failure to do so will have two effects -- on the very first round of processing some blocks will be processed while their input values are still undefined, and on all subsequent rounds delays will be introduced into the system, resulting in inaccuracies. Sequencing is necessary to avoid such problems.

The processing sequence may be manually specified, as in the case of the Bristol UCS3000 Process Controller [39], at compile time. This has two disadvantages -- it involves extra work for the user (and introduces another source of error), and it cannot cater for on-line changes to the control strategy.

While the function of sequencing may be left to the Supervisor in the run-time system, it is preferrably done at an earlier stage in order to alert the user to errors as early as possible in the software production process.

Sequencing would be a straightforward process if the inter-connection of blocks did not result in closed loops. All that is necessary would be to start with input interface blocks and gradually proceed through the inner blocks. For example, in Fig 5.1, A and E are

analog input blocks and D is an analog output block. There exist several possible sequences -- ABEFCD, AEBFCD, AEFBCD, EABFCD etc. -- all of which will produce identical results.

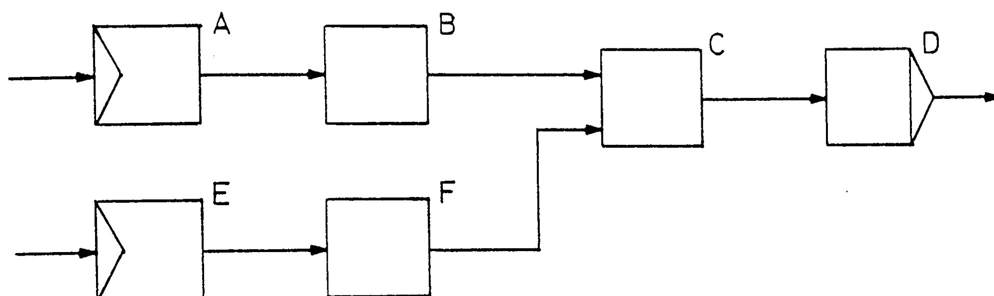
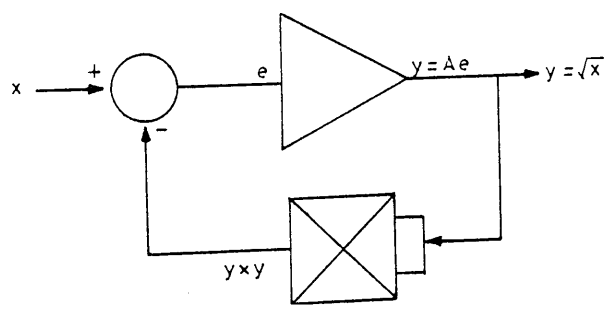


Figure 5.1 Example of Interconnection of Blocks

Often, however, the data flow between the blocks forms closed loops. Two situations are likely to arise.

In the first, an attempt is made to obtain an inverse function by feeding the output of a high gain block through a particular function block back into its input, a common practice in analog computing. Fig 5.2 shows a method of obtaining the square root function from a square



$A \gg 1$

$y = Ae = A(x - y^2)$

$x - y^2 = \frac{y}{A} \simeq 0$

$y = \sqrt{x}$

Figure 5.2 Square Root From Square Function

function. If all the blocks forming the loop have outputs which are functions of the instantaneous values of their inputs, an algebraic loop exists. While analog components can handle this configuration without any trouble since processing is carried out in a truly parallel manner, a

digital solution is more awkward, since it is equivalent to an implicit expression of the form y=f(x,y). Several continuous simulation languages (e.g. MIMIC, CSSL, CSMP [103-105]) permit such expressions provided they are declared to be IMPLICIT; solutions to these expressions can only be arrived at by an iterative process.

While such solutions may be acceptable in simulation programs, they are unacceptable in DDC applications as the number of iterations vary depending on the specific data involved and the accuracy required; moreover, convergence cannot be guaranteed. In such a case therefore, the compound function of the group of blocks must be realised by a single new functional block.

In the second situation, at least one of the blocks in the loop is retrospective (Section 4.2). Such a block effectively breaks open the loop. An example is an INTEGRATOR block using the rectangular integration formula

$$I_n = I_{n-1} + TU_{n-1}$$

The sequencing problem is solved by processing all retrospective blocks first, followed by the remaining non-retrospective blocks.

The question still exists, however, as to the order of processing among the retrospective blocks. When retrospective blocks are separated by non-retrospective blocks the processing sequence does not matter, but if the output of one retrospective block directly feeds the input of another, a problem arises. Consider for example two simple delay blocks A and B which delay the input value for one sampling interval, connected as shown in Fig 5.3, where C is an instantaneous function block. Assume that

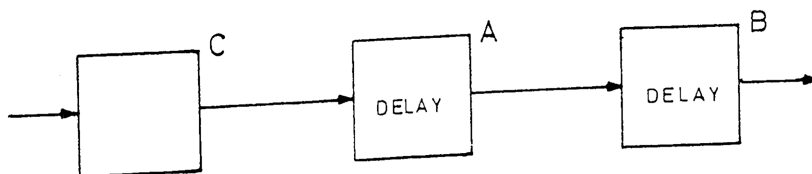
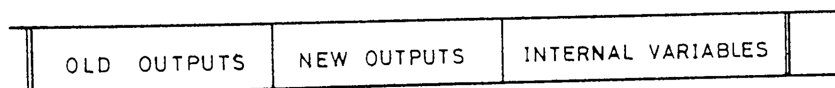


Figure 5.3 Delay Block Chain

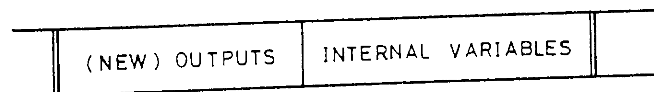
at time nT all the blocks have been processed, and the values of the outputs of blocks C, A and B are C_n, A_n and B_n. In the next round of processing, at time (n+1)T, if A is processed before B, its output becomes $A_{n+1} = C_n$. The result of processing B should be $B_{n+1} = A_n$, but the value of A_n has already been lost. This is a consequence of allocating storage to output (and internal) values only, and not to input values (see Section 5.4.1).

There are three solutions to this problem. The first requires processing of chained retrospective blocks in an order exactly reverse to that for non-retrospective block chains, such that the last block in the chain is processed first. This strategy however again fails if the retrospective blocks form a closed loop. A simple unity gain block may be inserted to break such a loop.

The second solution necessitates a distinction beween new and old values for the outputs of retrospective blocks (see Fig 5.4a). When such



(a) Retrospective Block



(b) Non-retrospective Block

Figure 5.4 Storage For Variables

a block is processed, the output results are stored in the 'new outputs' area of the block's data storage area. Non-retrospective blocks have only one set of output values -- 'new outputs' (see Fig 5.4b). The input of a non-retrospective block must always use the new values of outputs. The input of a retrospective block, if connected to another retrospective

block, must use the latter's old values. The distinction between 'new outputs' and 'old outputs' only exists in the Run-time Data Structure, and has no effect on the Graphic Data Structure. This method has several disadvantages :-

(i) extra storage is required for retrospective blocks, which must be present regardless of the existence of retrospective block chains;

(ii) the GPCL compiler has to differentiate between the two types of storage allocation when producing the Run-time Data Structure;

(iii) the same is required of the Supervisor when on-line reconfigurations are made;

(iv) if all inputs point to 'old outputs' of retrospective blocks, the 'old outputs' must be updated by the Supervisor to the new values before the non-retrospective blocks are processed;

(v) if the inputs of non-retrospective blocks point to 'new outputs', the 'old outputs' may be updated at the very beginning of each retrospective function routine, but this method complicates the setting of input pointers when making connections (although this can only done when the control loop is not running, and does not incur any run-time overhead).

The third solution avoids the problem altogether by prohibiting direct connections between retrospective blocks. If such blocks have to be connected, they must be separated by unity gain buffer blocks.

The second solution is too complicated, while the third is unnecessarily harsh. It appears that the first solution (sequencing) is

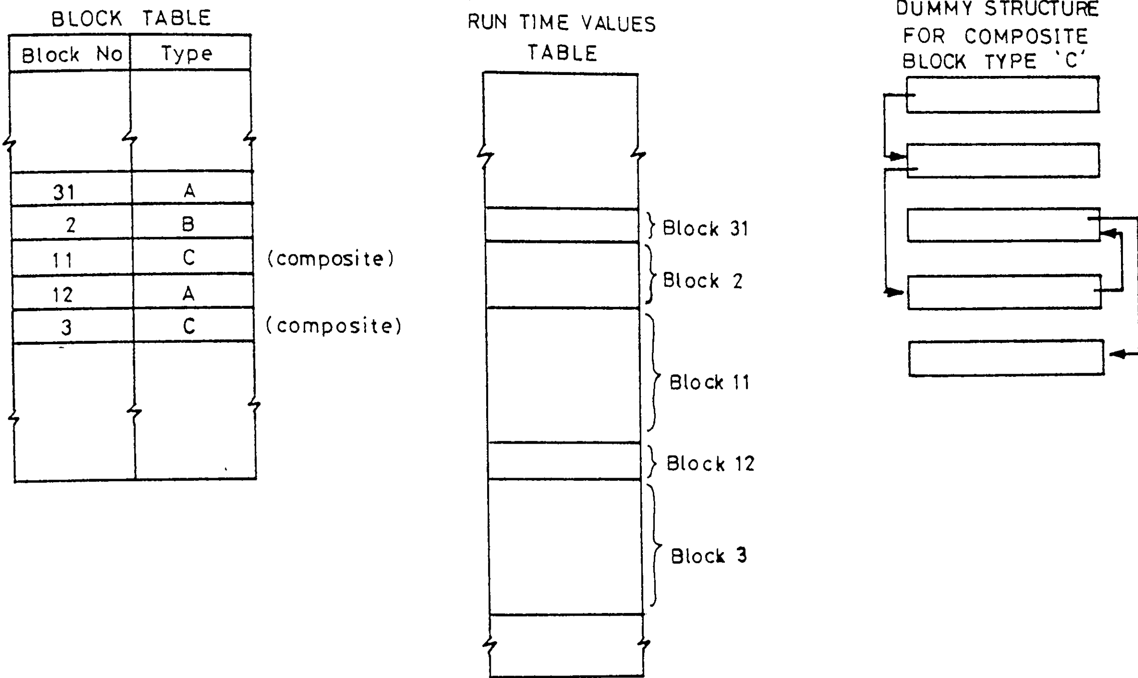
acceptable, with a restriction on retrospective blocks forming closed loops.

5.3.2 Treatment of Macros and Subpictures

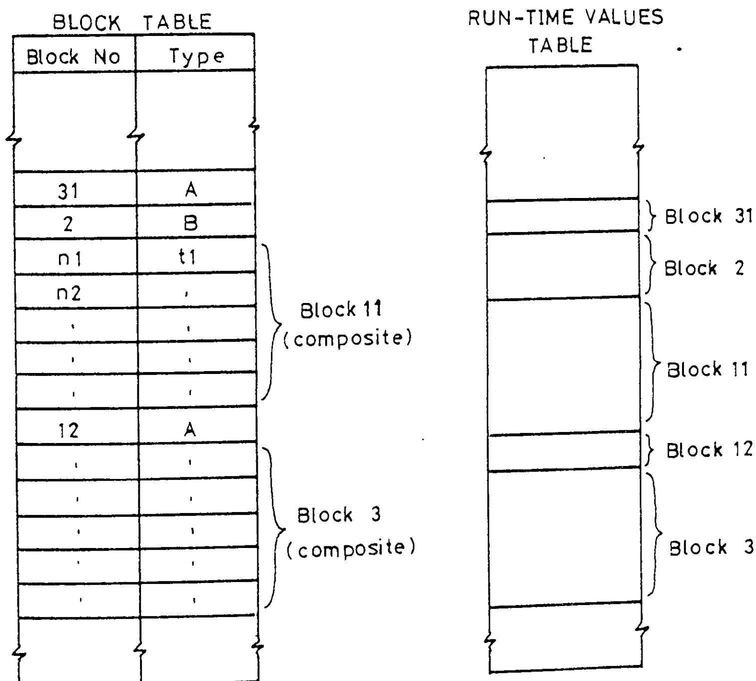
There are two ways of treating a composite block (i.e. an assembly of blocks to be treated as a single block -- Section 3.2.1). If they are looked upon as analogous to subroutines in normal programming languages, they will be treated as individual entities during execution and possess their own variables. They have no identity problem in the run-time system, and in common with all subroutine calls, little space in the Block Table (which defines the configuration, Section 5.4.2) is required for each occurrence of the subroutine. However, it also means that a dummy structure is required to define its internal composition, and variables storage in the Run-time Values Table (Section 5.4.1) must still be allocated for every block inside it, this storage being replicated for every occurrence of the subroutine block. Fig 5.5a shows conceptually the effect on the Run-time Data Structure.

Also, in common with all subroutine calls, more parameter passing is required, resulting in slower execution.

Sequencing is more difficult when the subroutine treatment is adopted. If the composite block has several outputs, all of which are affected by the instantaneous values of its inputs, it is a purely non-retrospective block; if none are affected, it is purely retrospective. In either case, sequencing has to be performed separately for each of the inner levels of a diagram. If the composite block has a mixture of retrospective and non-retrospective paths, sequencing becomes a non-trivial problem.



(a) 'Subroutine' treatment



(b) 'Macro' treatment

Figure 5.5 Effect of Composite Blocks on Run-time Data Structure (Simplified)

If composite blocks are treated as macros, they must be expanded down to the lowest level, resulting in only simple blocks. Similar to the expansion of macro code statements, this requires more storage since the structure of the macro is repeated for every occurrence of the macro block as shown in Fig 5.5b . The advantage lies in the simplicity of execution during run-time -- no dummy structures are required, and sequencing is done to the one and only level of (simple) blocks that is present.

However, unless a certain amount of redundant information is retained, the identity of each macro block will be lost. If the user has defined a frequently used network of blocks to be a macro block, he will be more interested in the block as a whole rather than its constituent elements. Also, reconfiguration in the form of deletions or additions of macro blocks is more awkward than for subroutine blocks because it involves multiple modifications in the Block Table (Section 5.4.2).

Since in the run-time system the emphasis is on execution efficiency of the control algorithms, the macro approach is preferred.

5.3.3 Generation of Control Program

For the GPCL compiler to generate the control program the user must supply both graphical and non-graphical data and a library of CORAL 66 procedures for the function algorithms (Fig 5.6). The user may also supply a list of values (initial values and constants) and text (block names and engineering units -- Section 3.3.5) for any block.

The functions of the GPCL compiler are :-

(i) expansion of macros and subpictures;

(ii) allocation of space for blocks and their variables in the various tables;

(iii) sequencing of the (expanded) blocks;

(iv) conversion of constants stored as strings in the Text Table into their numerical values for storage in the Run-time Values Table;

(v) inclusion of all CORAL 66 procedures required;

(vi) generation of the Run-time Data Structure in a machine independent format;

(vii) generation of listings and error messages.

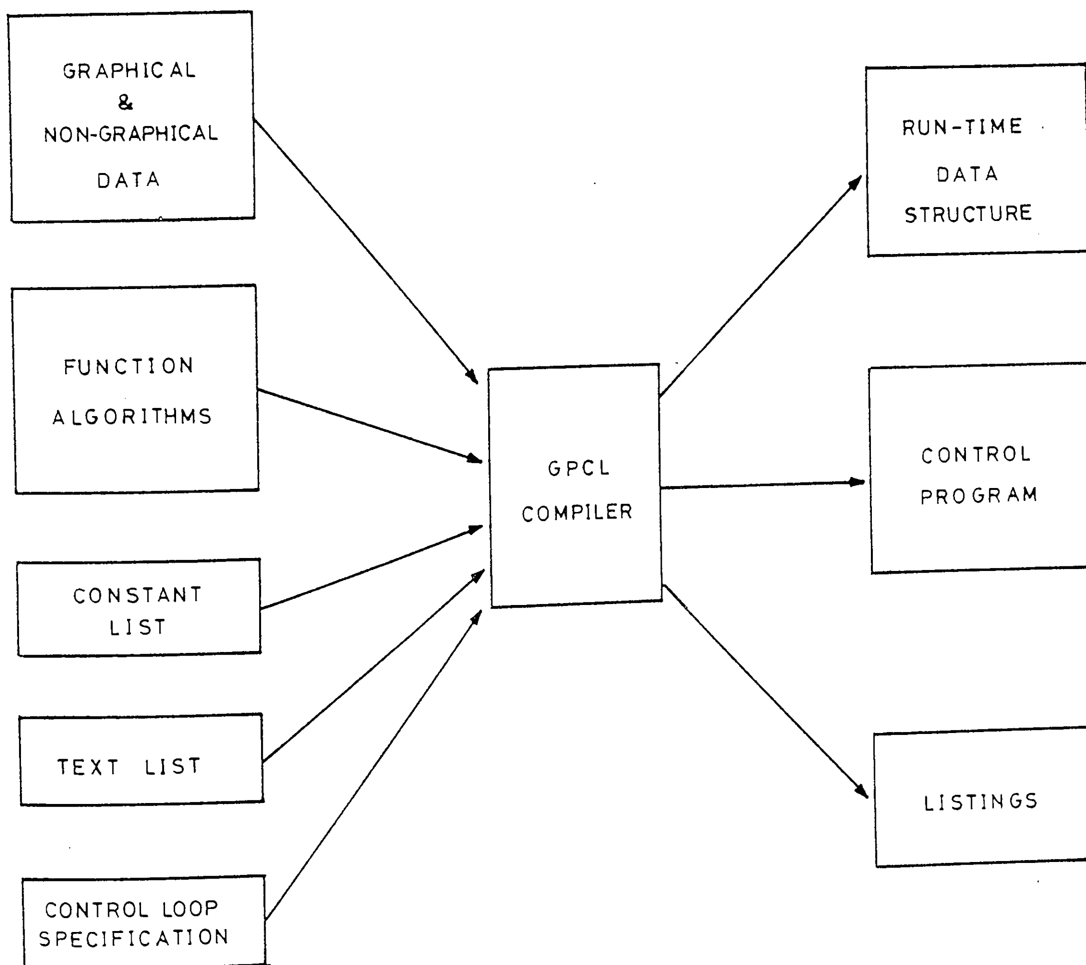


Figure 5.6 Generation of Control Program

5.4 Run-time Data Structure

The requirements of the Run-time Data Structure are to permit efficient real-time execution while at the same time preserving the structured feature of the original control diagram using the minimum of information possible.

The Graphic Data Structure is designed to facilitate graphic editing; it does not lend itself to efficient execution of the control algorithms. In addition, certain run-time elements are missing. Search procedures used to establish connections in the Graphic Data Structure are time-consuming. It is therefore necessary to define a Run-time Data Structure which optimises run-time execution.

The structure of the block diagrams must be preserved in the run-time data in order to permit on-line interrogation and modification, as well as to enable reverse compilation. Ideally, in the absence of memory constraints, an up-to-date Graphic Data Structure (and therefore diagram) should be recoverable from the Run-time Data Structure. This requires non-functional information like x-y coordinates, shape and random text to be kept in the Run-time Data Structure. If these are omitted only partial recovery is possible. This is not a serious drawback, since a copy of the original Graphic Data Structure must exist, and the reverse compiler can use it in conjunction with the Run-time Data Structure to produce a revised diagram which is logically and functionally correct, if not totally satisfactory from an aesthetic point of view.

Following this policy, the Run-time Data Structure consists of four tables :-

(i) a Run-time Values Table

(ii) a Block Table

(iii) a Type Table

(iv) a Text Table .

5.4.1 Run-time Values Table (RVT)

In the run-time system it is necessary to provide storage for the variables for each block. Three types of variables are involved -- input values derived from other blocks, output values to be accessed by other blocks, and internal variables.

Since the inputs of a block are derived from the outputs of other blocks it is neither necessary nor desirable to store both input and output values.

If output values alone are stored, a set of pointers is required to define input values, pointing to the values of the outputs to which they are connected (Fig 5.7b). This is not an overhead because it also defines the structure (interconnection) of the control diagram.

The alternative approach requires all input values which are connected to a block's output to be updated on completion of computation for that particular block. Since an output may feed several inputs, a slightly more complex pointer system has to be maintained (Fig 5.7c). Also, more values have to be stored and the updating of all the inputs involves extra processing.

It is concluded that storage of outputs is preferred to storage of inputs. For each block, all outputs and internal variables (if any) are stored in a contiguous area in the RVT which is a one-dimensional floating point array (Fig 5.8).

In both cases, instead of the pointers specifying the location of a block and its input or output number as is the case in the Graphic Data Structure, they point to the actual locations in the RVT where the value of the variable is to be found, thereby minimising the number of

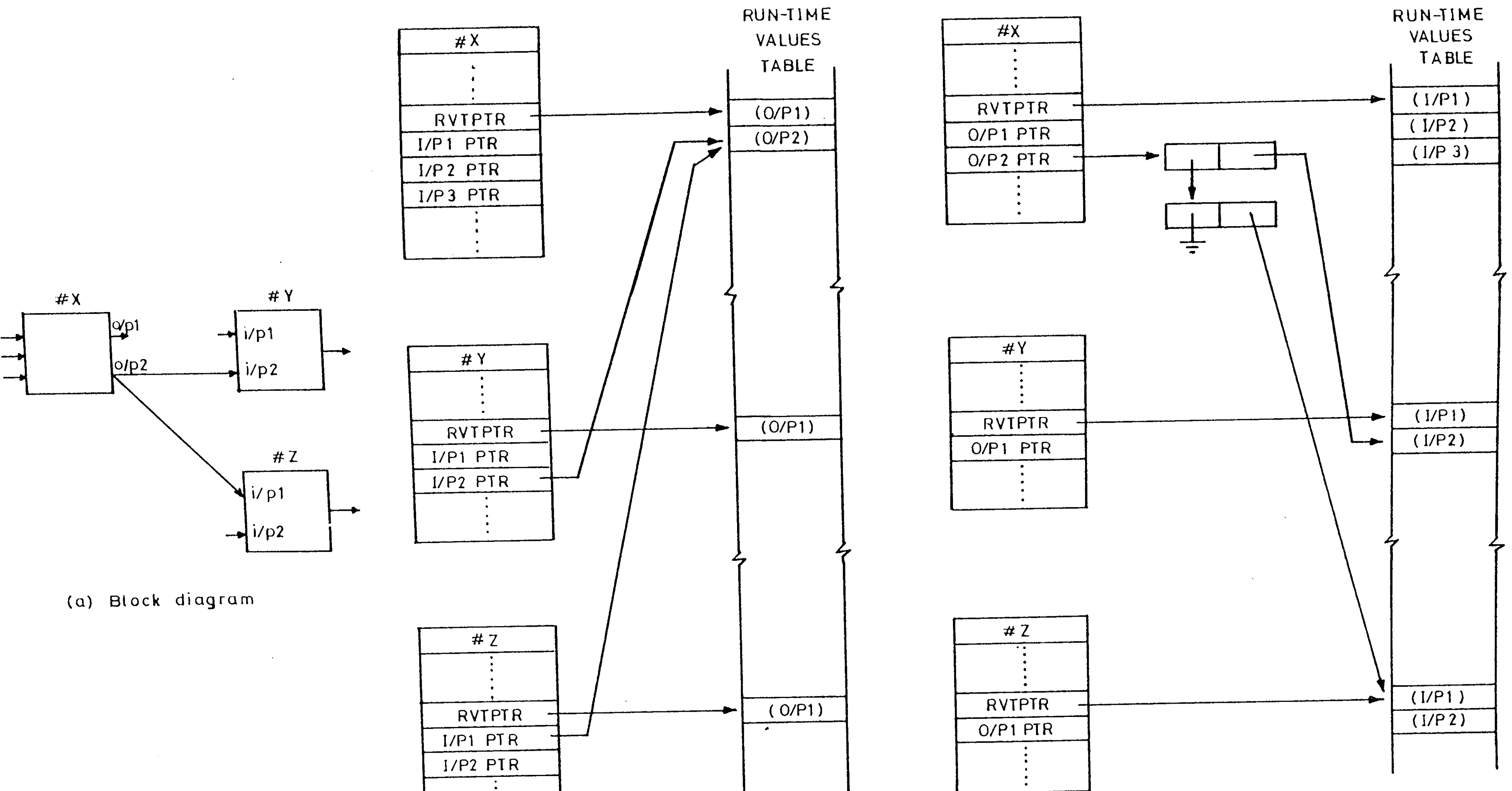


Figure 5.7 Storage of Variables -- Effect on Data Structure

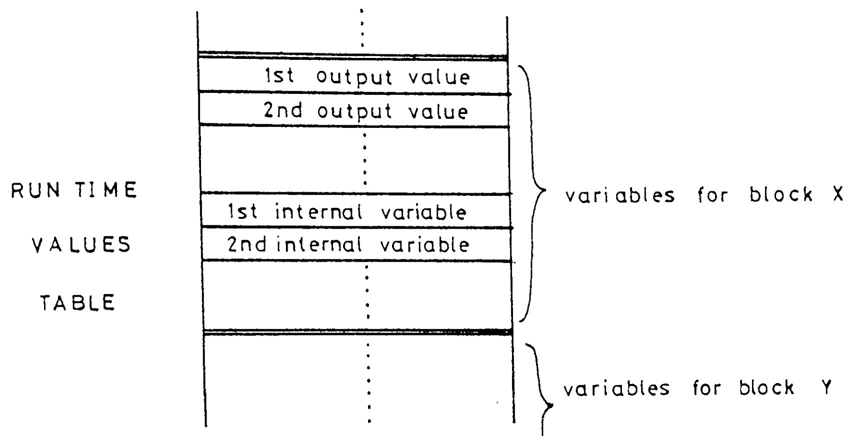


Figure 5.8 Run-time Values Table

references necessary to access the values. There is one drawback, however, as mentioned below.

While interconnection information is thus not explicitly defined, it can be deduced from a knowledge of the locations of the variables of each block in the RVT, when requested by the operator. The ease of obtaining interconnection information is sacrificed for faster access to variables during execution. This is perfectly acceptable since all operator interactions are low priority tasks. However, a consequence is that function block procedures are unable to determine the connection details. For example, if a LOGGER block exists and is connected to a block in order to log the latter's output, it cannot easily identify the variable that it is logging purely from the existing connection information.

5.4.2 Block Table

The Block Table defines the formal structure of the control loops. All composite blocks are expanded into their constituent blocks. Every block whether simple or composite has an entry in the Block Table. If composite blocks were simply expanded, their identity would be lost, resulting in a configuration of blocks conceptually different from the original process control diagrams. It is therefore necessary to retain

the identity of composite blocks. Only simple blocks have an associated algorithm, possess variables, and are executed. Composite blocks do not possess variables of their own, nor are they executed.

The Block Table consists of several linked lists of Block nodes, each list linking the blocks appearing in the same picture. Composite Block nodes are further linked to the list of its constituent blocks. Simple Block nodes and Composite Block nodes (Fig 5.9a,b) have largely similar data elements :-

(i) GLINK -- link to other block nodes

(ii) LNTH -- length of node

(iii) LLINK -- local link

(iv) local block number (LBLK)

(v) global block number (GBLK)

(vi) a pointer (TYPEPTR) to its TYPE description

(vii) a pointer (TXTPTR) to its specific text

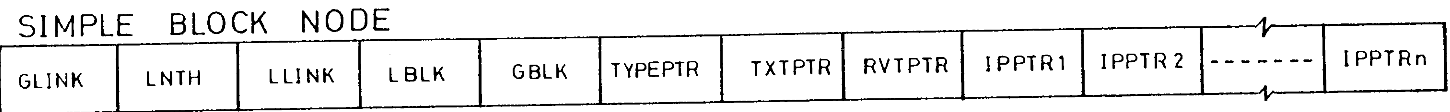
(viii) a pointer (RVTPTR) to its variables

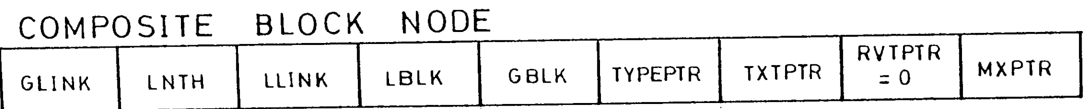
(ix) (for simple blocks only) a pointer (IPPTR) for each input.

In both Simple and Composite Block nodes, GLINK links all nodes. The processing sequence is determined by the order of linking. Composite blocks are not processed and their nodes appear after those of the simple blocks. As usual, the length of each node is stored. The LENGTH is always the length in (CORAL 66 integer) words.

The LOCAL LINK links together all blocks belonging to the same picture or composite block.

In the Graphic Data Structure, a block may be referred to by its block number. Since the block may belong to a composite block, more than one block may have the same number in the Run-time Data Structure because of the expansion of composite blocks. The block number is therefore only unique within its own picture, and is called the LOCAL BLOCK NUMBER. If a





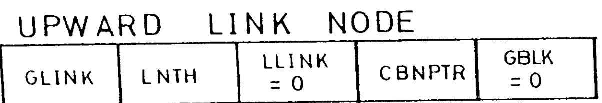
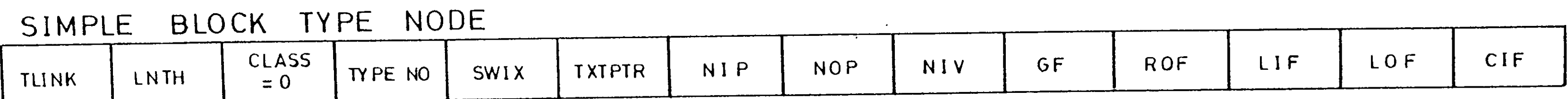


Figure 5.9 Simple and Composite Block Nodes



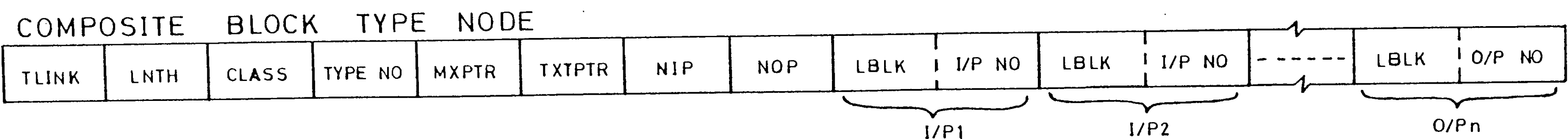


Figure 5.10 Simple and Composite Block Type Nodes

block constitutes part of a composite block which in turn belongs to another composite block, it can be specified in the run-time system by qualifying its Local Block Number with those of all intermediate levels.

It is also possible to identify the block by its GLOBAL BLOCK NUMBER, which is truly unique to each block and is assigned by the GPCL compiler. Both local and global block numbers are to be found in the Simple and Composite Block node records.

The TYPEPTR is a pointer to the appropriate TYPE node which contains type-specific information (Section 5.4.3).

The TXTPTR points to any block-specific text that may be associated with the block (Section 5.4.6).

Variables associated with composite blocks are not stored separately because these variables really belong to the constituent simple blocks. Separate storage for composite blocks would therefore be redundant and necessitate extra processing in updating the values. Hence only simple blocks possess variables in the RVT, and RVTPTR is the pointer to the start of each block's variables in the RVT. A zero value in the Composite Block node corresponding to the RVTPTR in the Simple Block node differentiates the two block nodes.

The remaining data in a Simple Block node define the interconnections -- for each input there is a pointer to the location in the RVT where the output value resides. As explained earlier, this minimises execution time.

In the Composite Block node, the MACRO EXPANSION POINTER (MXPTR) links it with its constituent blocks at the next level. This enables traversal from a higher to a lower level, but not in the opposite direction. It is possible to find a block by specifying the local block numbers of successive composite blocks higher up in the hierarchy, but not the reverse. An upward link is required for this purpose and is

provided by the Upward Link node (Fig 5.9c). It is always the last node in the list formed by the local links (all blocks belonging to the same picture), and contains a pointer CBNPTR to the Composite Block node which the group of blocks constitute.

5.4.3 TYPE Table

The TYPE table holds information specific to each block type. The description for a simple block TYPE is different from that of a composite block TYPE. Only simple blocks possess algorithms and real variables. Composite blocks are not executed since they are expanded into their constituent simple blocks.

The TYPE table is organised as a linked list, formed of two types of nodes, each node containing information for one block TYPE (Fig 5.10).

Each Simple Block Type node holds the following description :-

(i) TLINK -- links all TYPE nodes

(ii) LNTH -- the length of the node (constant)

(iii) CLASS, which is zero for simple blocks

(iv) TYPE number which identifies the block function

(v) SWIX -- the index to its procedure call

(vi) a pointer (TXTPTR) to type-specific text

(vii) the number of input terminals (NIP)

(viii) the number of output terminals (NOP)

(ix) the number of internal variables (NIV)

(x) a General Flag word (GF)

(xi) a Reset Outputs Flag word (ROF)

(xii) Logical Inputs and Logical Outputs Flag words (LIF,LOF)

(xiii) a Constant Inputs Flag word (CIF) .

The length of all Simple Block Type nodes is fixed; that of Composite Block Type nodes is not.

The TYPE number is used only during operator interaction and not during processing of the blocks.

The algorithm to be executed for a block depends on its TYPE. To process a block, the supervisor must call the appropriate routine to execute the desired algorithm. This may be done in various ways.

Selection using a nested IF statement, which compares TYPE numbers, is slow unless there is only a handful of TYPEs. A CASE statement, while basically performing the same function, does so more efficiently, but it is not available in CORAL 66.

A 'computed goto' statement may be used to select the correct procedure. In CORAL 66 the mechanism is called a SWITCH, which is an array of labels to which program execution may be transferred. Since the TYPE numbers are seldom consecutive, they may not be used directly as the index to the switch. Instead, a label is generated for each function procedure call and the index of the label within the SWITCH declaration is stored in the Switch Index (SWIX) word in the corresponding TYPE node. The Switch Index is used by the Supervisor to effect the correct procedure call (see Fig 5.11).

The CLASS of a block determines whether it is a simple block, a subpicture or a macro block, for which the CLASS values are 0, 1, and 2 respectively. This information is required during sequencing and execution -- only simple blocks are executed. It is also required during modification -- a subpicture can have its internal structure modified whereas a macro block cannot.

TXTPTR points to the Text node which contains type-specific text (function and terminal names).

```
'CORAL'
'PROGRAM' CALL BLOCK

'EXTERNAL' ('PROCEDURE' CALL BLOCK );
'EXTERNAL' ('INTEGER' SWIX ) ;

'EXTERNAL' ('PROCEDURE' AINPUT ) ;
'EXTERNAL' ('PROCEDURE' CINPUT ) ;
'EXTERNAL' ('PROCEDURE' ANOUT ) ;
'EXTERNAL' ('PROCEDURE' XYFUNCTION ) ;
'EXTERNAL' ('PROCEDURE' MULTIPLIER ) ;
'EXTERNAL' ('PROCEDURE' PID ) ;

'SEGMENT' ONE
'BEGIN'

'PROCEDURE' CALL BLOCK ;
'BEGIN'

        'SWITCH' PROCLABEL:= L1,L2,L3,L4,L5,L6 ;

        'GOTO' PROCLABEL[ SWIX ] ;

        L1:  AINPUT ;        'GOTO' EOR ;

        L2:  CINPUT ;        'GOTO' EOR ;

        L3:  ANOUT ;         'GOTO' EOR ;

        L4:  XYFUNCTION ;    'GOTO' EOR ;

        L5:  MULTIPLIER ;    'GOTO' EOR ;

        L6:  PID ;           'GOTO' EOR ;

EOR:

'END';

'END'
'FINISH'
```

Figure 5.11 Calling of Function Procedures Via SWITCH

The number of inputs determines the length of a Simple Block node and is therefore required when a new block is added to the system.

The number of outputs and internal variables determine the allocation of space in the RVT for each block.

The various flag words have been described in Section 4.2 . The GF and ROF words are used in the sequencing operation; the other flag words enable checking of connections made on-line, and provide an additional protection against human errors.

Composite Block Type nodes contain a different set of data :-

(i) TLINK -- links all TYPE nodes

(ii) LNTH -- the length of each node, which varies according to the number of terminals present

(iii) CLASS -- a value of 1 for subpictures and 2 for macros

(iv) TYPE number

(v) MXPTR -- a pointer to its expanded structure

(vi) a pointer (TXTPTR) to any type-specific text

(vii) number of inputs (NIP)

(viii) number of outputs (NOP)

(ix) input and output terminal equates .

TLINK, LNTH, CLASS and TXTPTR function in the same way as in Simple Block Type nodes. NIP and NOP are used in conjunction with the input and output terminal equates (see below) for operator functions.

Since connections and variables in the Run-time Data Structure are explicitly defined for simple blocks only (connections to a composite block are translated into lower level connections involving its internal blocks), it is necessary to include in the TYPE information a set of equates which relate the external terminals of the composite block to those of its internal blocks. For each input terminal there is a pair of numbers which specify the equivalent internal block and its input; output

equates are similar. This imposes a restriction of one internal input to each macro or subpicture input, as shown in Fig 5.12a,b . This is not a serious drawback as it may be overcome if necessary by the introduction of unity gain blocks to act as distributors (Fig 5.12c).

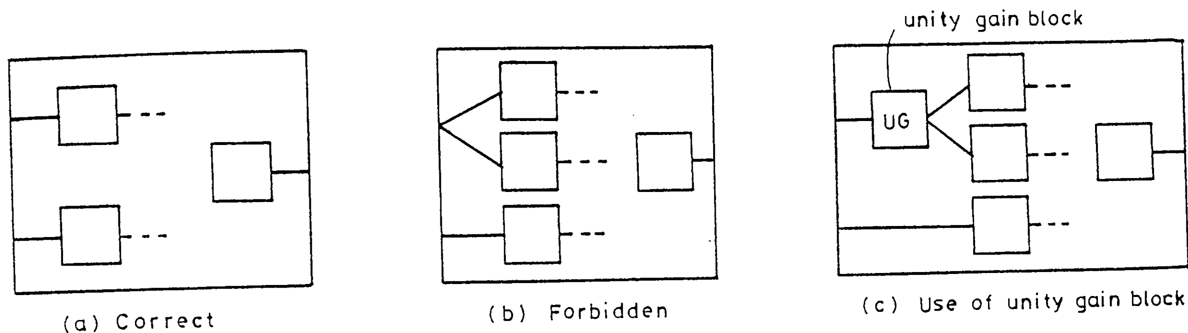


Figure 5.12 Composite Block Inputs

The length of a Composite Block Type node is therefore dependent upon the number of terminals it possesses.

MXPTR is a pointer to the internal structural description of the composite block. The use of a dummy structure is unnecessary since the structure is defined by any one of the existing composite blocks (of the same TYPE) in the Block Table. A consequence of this approach is the requirement that at least one copy of the macro or subpicture must exist.

Since composite blocks are not executed, all information related to execution do not exist. Similarly, information pertaining to connections (the various flag words) do not exist.

The components of the Run-time Data Structure dealt with so far are shown in Fig 5.13 .

5.4.4 Accessing of Variables

The format of a functional block's algorithm written as a CORAL 66 procedure has been discussed in Section 4.3 . This section discusses

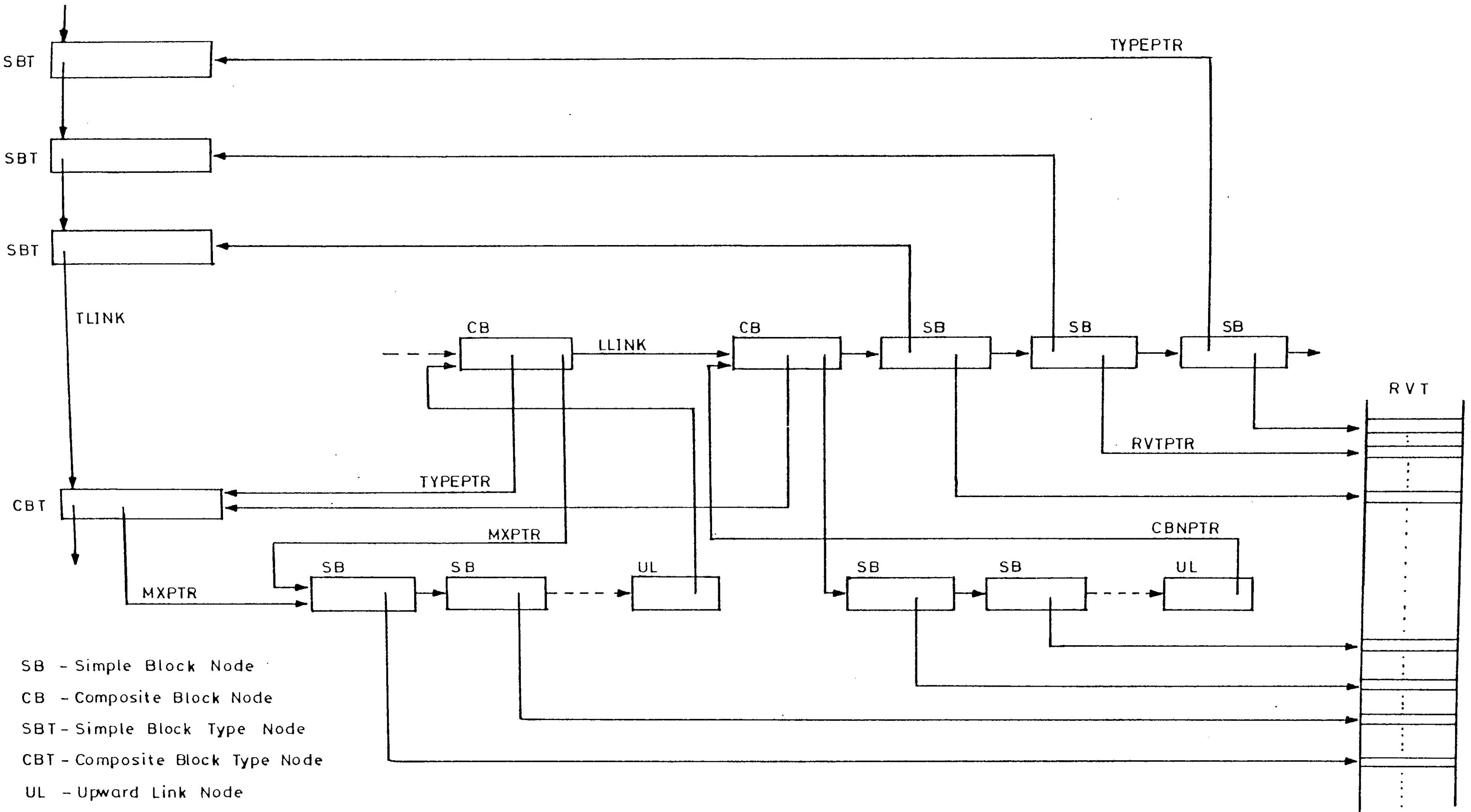


Figure 5.13 Main Components of Run-time Data Structure

details of the access mechanism.

In each procedure, two types of data accesses from the Run-time Data Structure are involved. The first type of access involves the internal variables and output variables which belong to the block being processed; the second type involves the block's input values which are in fact the outputs of other blocks. All these values are to be found in the Run-time Values Table (Section 5.4.1).

A procedure can obtain variables from the RVT in two ways. It can rely on the Supervisor to find and copy all its input values into an array called INPUT, and all its variables into another array called OUTPUT. Then the arithmetic or logical operations are performed on these two arrays. The values in the OUTPUT array are then copied back into the RVT by the Supervisor.

For example the equation for a MULTIPLIER block can be written as

OUTPUT1 := INPUT1 * INPUT2

Using the intermediate arrays, the equation would be

OUTPUT[1] := INPUT[1] * INPUT[2]

The advantages of this method are the ability to handle 'special cases' like JUNCTIONS (see Section 5.4.5), and the security it provides (the effect of any indexing errors are more likely to be limited to the intermediate arrays and therefore the block's own variables). The disadvantage is the extra processing overhead incurred in transferring to and from the intermediate arrays.

The second method involves accessing the RVT values directly through pointers. Before executing the functional algorithm for a block, the Supervisor obtains from the Simple Block node the pointer to its RVT entries (RVTPTR) and places it in a global variable called OPBASE, and places the address of the first input pointer in another global called IPBASE.

The first output of the block can then be accessed by the term

RVT [OPBASE] ,

the second output by the term

RVT [OPBASE+1]

and so on. Internal variables are accessed in a similar fashion.

The first input value of the block can be accessed by the term

RVT [RDS [IPBASE]] ,

the second input value by the term

RVT [RDS [IPBASE+1]]

and so on, where RDS refers to the array which contains all tables except the RVT (see Section 5.4.7).

Taking for simplicity the example of the MULTIPLIER block, the full expression relating its output to its two inputs is

RVT[OPBASE] := RVT[RDS[IPBASE]]*RVT[RDS[IPBASE+1]]

This is cumbersome and may lead to errors, especially if the algorithm is a more complex one.

One of the useful features of a CORAL 66 compiler is the incorporation of a macro processor, enabling definition of macro names in the form of identifiers to replace often lengthy blocks of code. By defining macros for each of the input and output terms, the expression can be written much more concisely (Fig 5.14). Not shown are the

```
'PROCEDURE' MULTIPLIER;

'DEFINE' OUTPUT "RVT[ OPBASE ]";
'DEFINE' INPUT1 "RVT[ RDS[ IPBASE ] ]";
'DEFINE' INPUT2 "RVT[ RDS[ IPBASE+1 ] ]";

'BEGIN'
      OUTPUT:= INPUT1*INPUT2
'END';

'DELETE' INPUT2; 'DELETE' INPUT1; 'DELETE' OUTPUT;
```

Figure 5.14 CORAL 66 Macro Definitions

'EXTERNAL' declarations for the arrays RDS and RVT and for the variables IPBASE and OPBASE.

The macro definitions need not be within the procedure body, as long as they appear before they are used. They may be kept separately and included at compile time by the CORAL 66 'LIBRARY' facility. A standard set may therefore be kept which relieves the engineer of the need to define them when creating a new algorithm.

Macros in CORAL 66 may also have parameters. Thus it is possible for example to have the following general macros :-

```
'DEFINE' OUTPUT(N) "RVT[ OPBASE-1+N ]";

'DEFINE' INPUT(N) "RVT[ RDS[ IPBASE-1+N ] ]";
```

in which case the MULTIPLIER algorithm becomes

```
OUTPUT(1) := INPUT(1)*INPUT(2)
```

This is useful when more variables are involved. Note that the variables may be called by any macro names provided the corresponding macro definitions exist.

5.4.5 Junction Blocks

If all graphical information was to be retained in the run-time system a problem would be posed by JUNCTION blocks (Section 3.3.2). The junction block was introduced as convenient means of segmenting a connection line in order to produce a neater diagram. Its role is therefore entirely graphical. Inclusion in the Run-time Data Structure has several undesirable effects, the most obvious being the additional storage required in the Block Table for every single junction.

If a junction is handled in exactly the same manner as other blocks, it must possess an output value, and a procedure which simply transfers the input value to its output -- that is, similar to a unity

gain block. This results in additional storage in the RVT and additional processing, both of which may be rather significant due to the fairly high proportion of junction blocks present. The procedure itself is too simple to occupy any significant storage.

To avoid the additional output value storage and processing, junction blocks must be treated differently. To obtain the input value of a block, if it is connected to a junction, the connection must be traced back through successive junctions if any until an ordinary block is reached, the output of the latter being the value sought. In this case the method of accessing input values by the function procedures described in Section 5.4.4 will no longer work. Every attempt to access an input value will require a test to see if the connection is to a junction. The execution overhead incurred will be even more than that for the first method. It is seen that the inclusion of junction blocks in the run-time system is not justified.

If junction blocks are omitted altogether from the Run-time Data Structure, the diagram loses it tidiness on reverse compilation (connection lines will be drawn by the shortest route), but it will not be rendered incomprehensible.

Since in the run-time system memory is restricted, graphical information should be omitted, and under such circumstances the retention of junctions is meaningless.

5.4.6 Text Table

The Text Table consists of a linked list of Text nodes, and is used to store 3 types of information :-

(i) function names and terminal names

(ii) block names and output engineering units

(iii) messages .

Function and terminal names are type-specific. Every function block type has a function name so that the function may be identified by name rather than by type number, in the run-time system. Similarly the presence of terminal names aids their identification. If they are absent, terminals have to be identified by terminal numbers.

The GPCL compiler concatenates the function name with the terminal names into a single text string and stores it in a Text node. Since each character is stored as 7 bits in an 8-bit byte, the 8th bit may be used to flag the last character of each name. This is denoted by the dots in the example of Fig 5.15 . If a terminal name is missing, it must still be indicated by a blank character.

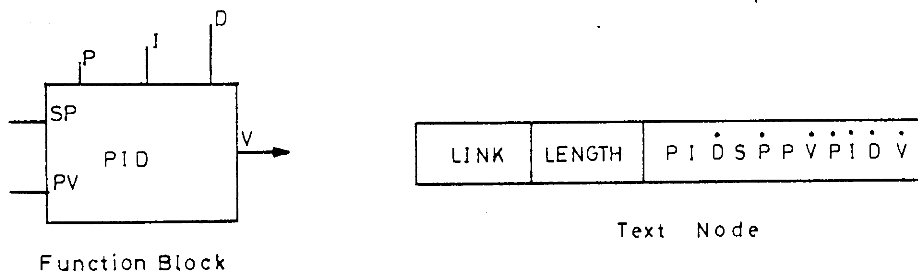


Figure 5.15 Function and Terminal Names

Also, every block may be given a name, and its outputs may be given engineering units, if required. As mentioned in Section 3.3.5, the current Graphic Data Structure does not cater for individual text within macros, so these have to be supplied separately during compilation. The block name and engineering units are concatenated into a single text string, as in the case of function and terminal names, and this is stored in another Text node as illustrated in Fig 5.16 . If the block has not been given a name nor any engineering units, then no Text node is created for that block.

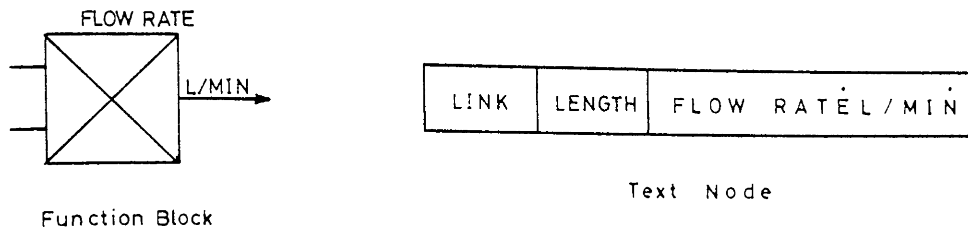


Figure 5.16 Block Name and Engineering Units

Text nodes also provide storage for messages which may be accessed by a suitably implemented MESSAGE block.

Each Text node is stored within the integer array RDS. The LENGTH is always given as the number of integer words (as determined by the CORAL 66 compiler) required to store the node. Thus for 16-bit integer representation, a Text node which contains a string of 10 characters (each stored in one byte) will have a length of 7.

Since this obviously affects the Run-time Data Structure, the GPCL compiler must know the integer representation used by the CORAL 66 compiler for the target machine. However, it is safe to assume a 16-bit integer representation.

5.4.7 Machine Independent Representation of RDS

All nodes described in the preceding sections are stored in a single integer array. The RVT is stored in a floating point array.

The Run-time Data Structure produced by the GPCL compiler is machine independent. This is possible because all pointers and links are array indices instead of absolute machine addresses. The output generated by the GPCL compiler is a list of integer and floating point array elements, in a decimal format (Fig 5.17). This output will subsequently be read by the Supervisor into the run-time system.

[RDS FLAG] [INDEX] <INT.VALUE> <INT.VALUE> .... [EOR]

[RVT FLAG] [INDEX] <FLT.VALUE> <FLT.VALUE> .... [EOR]

Figure 5.17 Format of RDS Output By GPCL Compiler

CHAPTER 6

RUN-TIME SYSTEM

6.1 Introduction

After the Supervisor and function block algorithms have been compiled by a CORAL 66 compiler, the resultant code may be linked with any machine dependent code modules. The linked code can then be loaded into the process computer for execution.

During execution, the Supervisor has to perform a number of tasks. The basic Supervisor functions are :-

(i) initialising the Run-time Data Structure;

(ii) sequencing the function blocks;

(iii) periodic processing of the function blocks;

(iv) operator interaction.

Other useful functions include :-

(i) logging;

(ii) alarm handling;

(iii) performing system integrity checks;

(iv) remote communications.

Each function may involve one or more tasks. Tasks operate at different levels of priority. Block processing is a high priority task. Operator interaction on the other hand involves low priority data structure interrogation and static display tasks, as well as a higher priority dynamic display task.

Details are given in the following sections.

6.2 Run-time Data Structure Initialisation

As stated in Section 5.4.7, the Run-time Data Structure produced by the GPCL compiler is in the form of a list of the contents of integer and floating point array elements. This representation enables it to be machine independent. The first function of the Supervisor is therefore to initialize the Run-time Data Structure by reading in these values from some external storage device. The format of the data was given in Section 5.4.7 .

6.3 Function Block Processing

Function block processing is the main high priority task. All low priority tasks are executed only when all block processing has been completed and before the next round of processing. Several control loops may be handled by the process computer. Loops may be on-line (active) or off-line. Each loop has its own processing interval, which must be a multiple of the basic processing interval. This number is stored in a counter. Processing is initiated by a real-time clock interrupt; if the start of the next basic processing interval has been reached, the Supervisor decrements the counter for each loop. When the counter is decremented to zero, it is reset, and the associated loop will be processed.

For each loop the Supervisor processes each function block in turn by calling its associated procedure. The processing sequence is determined by the order in which the Block nodes are linked in the Run-time Data Structure. Only simple blocks are processed, since composite blocks are expanded into simple blocks by the GPCL compiler. Retrospective blocks are processed first, since by definition the current output values may be computed without knowledge of the current input

values. Then the new output values of the remaining blocks may be computed. The order of processing is :-

(i) retrospective blocks

(ii) input interface blocks

(iii) non-retrospective blocks

(iv) output interface blocks .

The Supervisor makes several variables available to the procedure being called. These are :-

(i) the absolute time

(ii) the loop's processing interval

(iii) a pointer to the start of the block's Run-time Values

(iv) a pointer to the block's first input descriptor (in its Block node)

(v) a pointer to the block's TYPE node .

As described in Section 5.4.3, the procedure is called using its Switch Index value in the TYPE node.

6.4 Sequencing

Sequencing is performed by the GPCL compiler in the course of the detection of errors such as algebraic loops as described in Section 5.3.1 . The Run-time Data Structure that is initially loaded into the run-time system is therefore already sequenced. However, when the control configuration is modified as a result of additions or deletions of blocks, or altered connections, the sequence will be upset. It is then necessary for the Supervisor to re-sequence the blocks. The sequencing algorithm is identical to that used by the GPCL compiler. The order of processing is determined by the order in which the Block nodes are linked. There is no necessity to provide a separate sequence table. The

sequencing algorithm is described below.

To illustrate the problem, the arbitrary configuration in Fig 6.1 is used. A, F are input interface blocks and J, H are output interface blocks. D and E are retrospective blocks; the rest are not.

Assume that originally the blocks have not yet been sequenced and are linked in a random order, for example as in Fig 6.2a. First a repeated search is made for any retrospective blocks, which will be moved to the head of the linked list by swapping links. The result is shown in Fig 6.2b.

The retrospective blocks then have to be sequenced, as mentioned in Section 5.3.1 . If D and E were not connected together their order would not matter. In this case however, E must be processed first. The inputs of D and E are checked for connection to a retrospective block. Since the input of E is connected to D, E must be moved in front of D in the linked list, resulting in Fig 6.2c.

After all retrospective blocks have been sequenced, the remainder of the list is searched for input interface blocks, which are moved to a position after the retrospective blocks (Fig 6.2d). The order amongst them is immaterial since they may not be connected together.

The remainder of the list is now searched for non-retrospective, computational blocks whose inputs are all defined -- that is, connected to blocks which are already sequenced. Sequenced blocks are indicated by their position in the linked list; alternatively it is possible to flag these blocks for example by temporarily negating their global block numbers in their Block nodes, or even better, to mark their Run-time Values by multiplication with a big value. In this case the next block to be sequenced is G (Fig 6.2e). Repeated searches will result in all non-retrospective blocks being sequenced (Fig 6.2f).

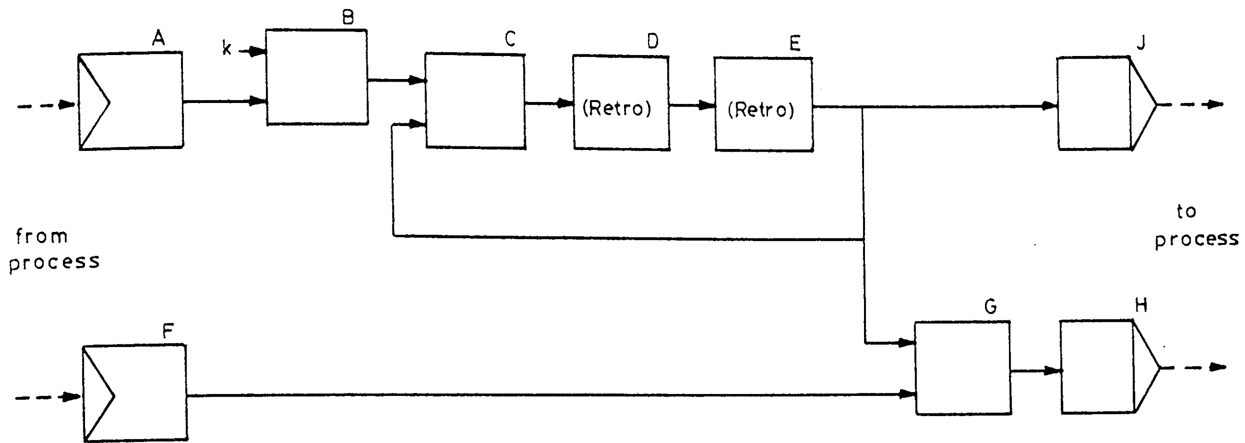
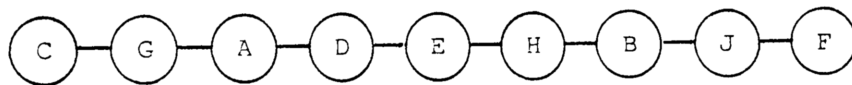
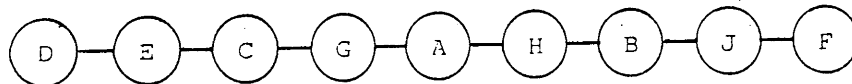


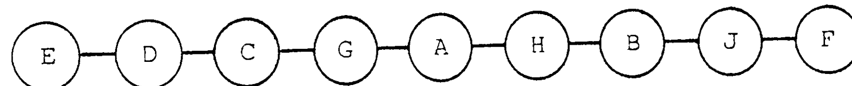
Figure 6.1 Example of Sequencing Problem



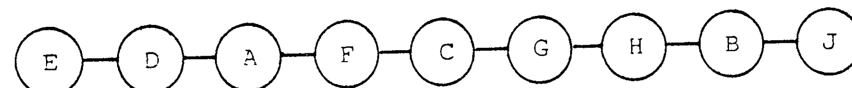
(a) Original sequence



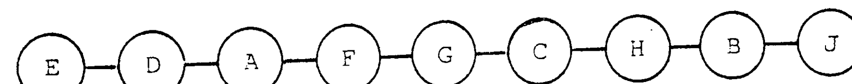
(b) Retrospective blocks moved



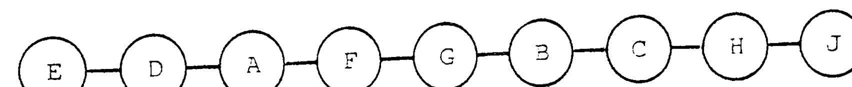
(c) Retrospective blocks sequenced



(d) Input interface blocks sequenced



(e)



(f) Sequencing complete

Figure 6.2 Intermediate Stages of Sequencing

The remainder of the simple blocks in the list will be output interface blocks among which the processing order again does not matter.

Detection of illegal loops is very simple. Each search through the unsequenced portion of the linked list should produce one block which may be added to the sequenced portion of the list. If either an algebraic loop, or a loop formed by retrospective blocks, exists this condition will ultimately fail.

It is apparent that sequencing is a very lengthy process. However, since sequencing need only be performed while a control loop is off-line (it may not be reconfigured when active), speed is of no consequence. Sequencing is a low priority task.

6.5 Operator Interaction

The main operator interactions are :-

(i) data display -- display of process variables, setpoints, control loop configuration etc.

(ii) data input -- modification of setpoint and other parameter values, and of control loop configurations.

Apart from the display of process variables, these are mainly low priority tasks, which execute only when the computer is not processing the control loops.

The man-machine hardware interface should preferably be a Visual Display Unit (VDU) with absolute cursor addressing which allows any process variable displayed to be periodically updated. Operator process panels with only LED alphanumeric displays are too restrictive for the amount of information that may be extracted from the system.

The Run-time Data Structure contains all the information necessary to allow the Supervisor to display the following details for each block

(not necessarily simultaneously) :-

(i) its global and local block numbers

(ii) the name of the function

(iii) the name of the block, if any

(iv) the name of each terminal

(v) the values of each input and output

(vi) the engineering units, if any, for each output

(vii) the values of its internal variables

(viii) the blocks to which it is connected

Items (iii) and (vi) will not always be displayed since only blocks of interest may have been given names and units.

The block may be specified via its global block number or its lineage (the list of intermediate blocks to which it belongs).

The display of most of the above information is straightforward. The display of input and output values has to be updated periodically (after every round of processing). The extraction of connection information from the Run-time Data Structure, however, involves multiple searches and comparisons -- a very time consuming process, but acceptable as it is a low priority task.

The Supervisor also allows modifications to the Run-time Data Structure. Most variables may be modified. This may be achieved via numeric entry, or via a gradual adjustment from the original value to avoid discontinuities, which is necessary if the value belongs to a loop that is active.

When a loop is off-line, it may be reconfigured -- connections may be modified and blocks added or deleted. Certain restrictions apply -- for example, macros may not be reconfigured, but their constants may be changed.

Additions and deletions of blocks involves a fair amount of processing. If a simple block is deleted, all connections to it must also be deleted. It is probably preferable to prevent blocks being deleted while connections to it exist, to force the operator to explicitly remove all connections before deleting the block. The space its variables occupy in the Run-time Values Table (RVT), as well as any text it possesses, must also be released.

The deletion of a macro block also results in deletion of all its constituent blocks. Since the structural definition of each macro block is implicit in the connections, at least one copy of each type of macro block, with all its RVT entries, may not be deleted.

Additions involve allocating Block nodes and space for storage of variables in the RVT. Also, some blocks may have variables which have to be initialised by the operator. Addition of macro blocks is particularly complicated, as it involves the creation of its constituent blocks.

6.6 Additional Supervisor Functions

The Supervisor functions may be developed to any degree of sophistication conceivable. Some useful facilities include logging, alarm handling, performing system checks, remote communications etc.

Logging and alarm handling may be implemented within the control loops as special function blocks. However, problems arise when attempting to log variables of a block withn a macro, for reasons similar to those stated in Section 3.3.5 ; this also applies to alarm handling. These two functions are therefore best performed by the Supervisor. Blocks whose outputs are to be logged or alarm monitored should be given block names to enable easy identification, and engineering units should also be provided.

Logging and alarm information may be stored in a LOG TABLE and ALARM TABLE (Fig 6.3). Some of these may be initialised at compile time,

LOGGING INTERVAL	BLOCK PTR	O/P NO.

(a)

BLOCK PTR	O/P NO.	LOW LIM.	HIGH LIM.

(b)

Figure 6.3 (a) LOG Table (b) ALARM Table

but the Supervisor should permit on-line specification of new variables to be logged or alarm-monitored.

The alarm handling function may vary somewhat in strategy. If variables are monitored every processing interval, the processing interval must be reasonably short to prevent an alarm condition from getting out of hand. An alternative is monitor these variables more frequently than the loop processing.

6.7 Interrupt Processing

The three main causes of interrupts are :-

(i) real-time clock

(ii) character input

(iii) character output .

The real-time clock interrupts update the system clock and enable the Supervisor to start processing the control loops at the correct times.

Character input/output interrupts are associated with communication functions.

Generally, input and output interface blocks do not use interrupts. Data acquisition and output from and to the process must be achieved within the processing of the block. There is therefore a limit on the time available to obtain each input or set up each output. In most cases this is not a problem, since even analog-to-digital converters are available which are not significantly slower than interrupt service routine overheads. If a particular input device responds too slowly, a possible solution is to initiate the data acquisition by the explicit use of a digital output block in the control loop, so that the data will be available at the next round of processing. Interrupt mechanisms are very machine dependent. Vectored interrupt mechanisms allow the processor to determine the source of the interrupt easily, but the number of vectored interrupts is limited. To resort to the time-consuming process of polling to discover the interrupting source would defeat the primary purpose of using interrupts (for the input/output interface blocks). The alternative is to use additional interrupt hardware.

Counter-type input functions which count the number of random pulses received are best implemented with hardware counters rather than with interrupt driven software counters. Input values should remain fixed for the whole duration of processing. Also, since processing of the loops is synchronous, little advantage is to be gained from the knowledge that a counter has reached a certain value, say, in the middle of processing because no action may be taken until the next round of processing.

Communications devices such as the UART (Universal Asynchronous Receiver Transmitter) are an exception. Communications is the only task invoked by a function block which may be executed asynchronously and proceed as a background task until completion.

CHAPTER 7

SYSTEM PERFORMANCE

7.1 Graphics

The software for the system has only been partially written. The Graphic Editor is capable of performing the basic editing functions. The size of the editor so far is approximately 20K (16-bit) words, but this does not provide a good indication of the memory requirements because it is dependent upon the level of sophistication sought, as well as the use of overlay techniques to minimise core requirements.

Only a limited number of function block types have been created. These include the ones used in the dye mixing example of Fig 3.3 . The development of the graphical and functional characteristics of each block type are given in the Appendix. The sizes of the Graphic Information node and the type-specific text nodes for each block type, and the size of each block node, in the Graphic Data Structure for the function blocks used are shown in Table 7-1. Values are given in 16-bit words.

The exact storage requirements for the block diagrams depends on the number of constants and user-defined text present in each block, as well as the amount of plant symbolic graphics included. Table 7-2 shows the total amount of data required for the various types of information to represent the block diagrams. Type-specific information occupies a substantial proportion of space in this example, but as the number of diagrams and blocks increase, the 'picture' requirements will increase proportionately whereas the increase in type-specific data will depend only on the number of different block types used.

Block Type	Graphic Info Node	Text Node†	Block Node
Analog Input	45	21	14
Counter Input	45	27	14
Analog Output	45	26	15
X-Y Function	37	6	10
Multiplier	33	25	12
P-I-D	45	36	18
Junction	15	0	10
Cascade (macro)	31	31	16

† type-specific text

Non-Graphical Data Node = 8 words each

Table 7-1 Individual Graphics Storage Requirements

	Type-Specific	Picture
Function Blocks	470	270
Graphic Blocks	150	100

Table 7-2 Total Graphics Storage For Block Diagram

7.2 Run-time System

The various memory requirements in the run-time system are shown in Table 7-3, where the column with the heading 'TYPE INFO' gives the total amount of data used to store type-specific information. The values of the Run-time Values Table requirements for each block takes into account the space required for constant parameter inputs. If some non-parameter inputs are connected to constants the total RVT requirements will increase. The RVT requirements assume the use of 2 words per floating point number. The X-Y function block holds 17 data points.

The Run-time Data Structure requirements for the dye mixing example total approximately 400 words of which 200 words are type information and 200 are individual block information and variables storage.

Figures related to execution are given in Table 7-4. On the Texas 990/10 minicomputer the control loop in the example takes approximately 28 msec to execute. This mediocre performance is due to the absence of floating point instructions and the relatively slow instruction execution times (between 1.5 and 20 microseconds) of this minicomputer. Floating point numbers are stored with 20 bits precision and a range of 10^{+76} to 10^{-76}. Floating point arithmetic accounts for a substantial proportion of the total processing -- addition of two floating point numbers takes an average of between 450 and 650 microseconds; division takes between 900 and 1000 microseconds. With the use of a faster CPU and hardware floating point, performance will improve substantially.

It is obvious that figures related to execution are dependent upon the specific computer used, whereas figures related to the data structure only depend on the design of each block and are largely machine independent (assuming the use of identical word lengths).

Block Type	Type Info	Block Node	RVT Size
Analog Input	22	11	8
Counter Input	23	11	4
Analog Output	22	12	6
X-Y Function	21	9	36
Multiplier	22	10	2
P-I-D	24	13	6
Cascade (macro)	40	9	0

Table 7-3 Run-time Data Storage Requirements

Procedure	Size	Approx.Time†
Analog Input	65	1.8
Counter Input	67	1.8
Analog Output	90	2.8
X-Y Function	162	4.2
Multiplier	36	1.0
P-I-D	137	6.8
Cascade (macro)	NA	NA

† in milliseconds

Table 7-4 990/10 Procedure Sizes and Execution Times

CHAPTER 8

CONCLUSIONS

This research has involved the investigation and design of a graphical block diagram approach to the generation of software for process control. A graphical language (GPCL) allows a process control block diagram to be created interactively at a graphics terminal, using the Graphic Editor. The use of the language requires no knowledge of formal programming, when working within the set of available function blocks. A novel feature is the inclusion of plant symbols and text in the block diagrams for purely commentary purposes. The uniformity of treatment for function blocks and graphic blocks (plant symbols) simplifies the Graphic Data Structure. The language also accomodates several different types of text (block, function and terminal names, constants and engineering units). Constant settings and connections are treated using a unified strategy. This simplifies the Graphic Data Structure as well as increases the flexibility of the system.

A macro and subpicture facility has been included to facilitate the repeated use of groups of blocks which collectively form compound functions and to enable the functional structuring of block diagrams. The current Graphic Data Structure does not cater for the presence of differing constants and text within similar macro blocks, but gains in terms of simplicity.

The salient graphical and functional attributes of function blocks have been identified and parameterised, enabling new block types to be accomodated easily. The creation of a new block type involves the graphical definition of the block, and in the case of function blocks the

specification of certain non-graphical properties as well as the supply of a CORAL 66 procedure, this being the only time a knowledge of programming is required.

The graphical nature of the system, together with the use of functional names for each terminal, reduce the chances of human error during creation of the diagrams. For example it is impossible for the user to include in the diagram a non-existent function block type, or make a connection to a non-existent block or terminal. During editing of the block diagram the GPCL editor is also able to perform a number of validity checks to eliminate errors such as connection between incompatible terminals. In this way the graphical feature is exploited to remove many of the sources of error likely with purely textual programming languages.

The completed process control diagram is submitted to the GPCL compiler, together with a library of CORAL 66 routines and any preset data; from these the GPCL compiler is able to generate a run-time system also in CORAL 66. This comprises a suite of machine-independent programs and a machine-independent Run-time Data Structure. These programs may then be compiled by a CORAL 66 compiler into machine dependent code for execution in the target process computer. The analysis by the GPCL compiler is facilitated by the function block approach, and also by the amount of preliminary checking already performed by the graphic editor. The generation of the Run-time Data Structure involves the creation of tables which define the structure of the block diagram, the expansion of macros and subpictures, the allocation of space for variables and parameters for each function block, the inclusion of type information for each block type, and the determination of the correct processing order for the blocks (sequencing).

The Run-time Data Structure permits identification of all variables and parameters for logging and user modification. Connection information is also retained so that the user may modify the block diagram in the run-time environment. Modification may result in a necessity to resequence the blocks, which is performed automatically.

All composite blocks are expanded into their constituent simple blocks so that no execution overhead is incurred by parameter passing. The identity of each composite block is retained, however, thus preserving the structural representation of the original block diagram. The organisation of the Run-time Data Structure allows accessing of variables by the function block procedures to be achieved directly through pointers. This maximises the real-time performance of the system since no searching is required in repetitive routines. The disadvantage is the amount of searching necessary for operator-related tasks, which is a small penalty since they are not executed frequently.

The further development of the system could take a number of directions depending on the application area anticipated. Several aspects have been incompletely developed; these include the modification of the Graphic Data Structure to handle differing text within similar macro blocks, the development of the GPCL compiler (which mainly performs data structure manipulation) and the run-time Supervisor. The role of the Supervisor in the initialisation of variables within the control algorithms and 'auto-manual bumpless transfer' will require special consideration.

The capabilities of the language have not been fully developed. The data structures describe data flow between function blocks. Although each connecting line has been used to carry one signal from one block to another, it is possible for each line to carry several signals. This is

analogous to the passing of arrays as procedure parameters in textual programming languages. The use of such a feature would be to complement the composite block facility, for the purpose of simplifying block diagrams. If this is done, extra information will be required to describe each terminal and enable detection of illegal connections.

APPENDIX

This section shows the development of the various components which constitute the block diagram of the dye mixing process shown in Fig 3.3, leading to the final Run-time Data Structure. The Analog Input and Cascade Controller blocks will be used to illustrate the details of some of the various components of the data structures.

A.1 Analog Input

Fig A.1 shows the graphical details of the Analog Input block. The coordinates of each point are given in GDU's (Graphic Display Units) -- the screen of the Tektronix 4051 terminal is divided into 1024 GDU's horizontally and 730 GDU's vertically.

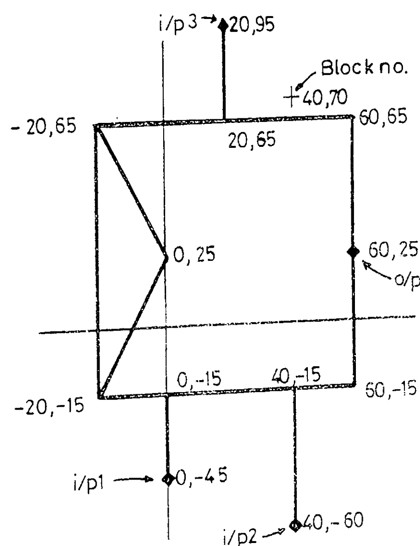


Figure A.1 Graphical Details of Analog Input Block

Graphic Information Node:

Type No. = 10

Class = 0

NIP = 3

NOP = 1

Block No. coords = (40,70)

I/O coords = (0,-45), (40,-60), (20,95), (60,25)

Picture coords = (-20+2000,-15), (-20,65), (60,65), (60,-15),

(-20,-15), (0,25), (-20,65), (20+2000,95),

(20,65), (0+2000,-15), (0,-45), (40+2000,-15),

(40,-60)

Non-Graphical Data Node:

NIV = 0
GF = 1
ROF = 0
LIF = 0
LOF = 0
CIF = 7

Type-specific Text Nodes:

↓	6	2000	0	A I N
↓	5	2000	1	S
↓	5	2000	2	Z
0	5	2000	3	A

Block Node:

	LNTH = 14	...	i/p1	i/p 2	i/p3	o/p

Run-time Data Structure:

Simple Block Node:

GLINK	LNTH =11	LLINK	LBLK	GBLK	TYPEPTR	TXTPTR	RVTPTR	IPPTR1	IPPTR2	IPPTR3

Type-specific Text Node:

LINK	LNTH = 6	A I Ṅ Ṡ Ż Ȧ ·

A.2 Cascade Controller

Fig A.2 shows the schematic of the Cascade controller block and its internal components.

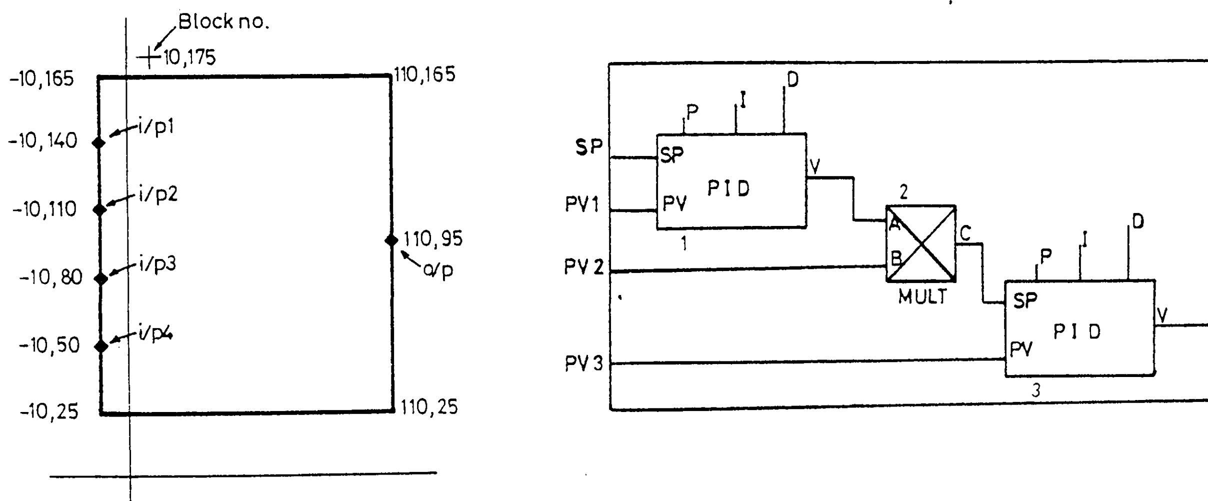


Figure A.2 Graphical Details of Cascade Controller Block

Graphic Information node:

TYPE no.= 100

Class= 2

NIP = 4

NOP = 1

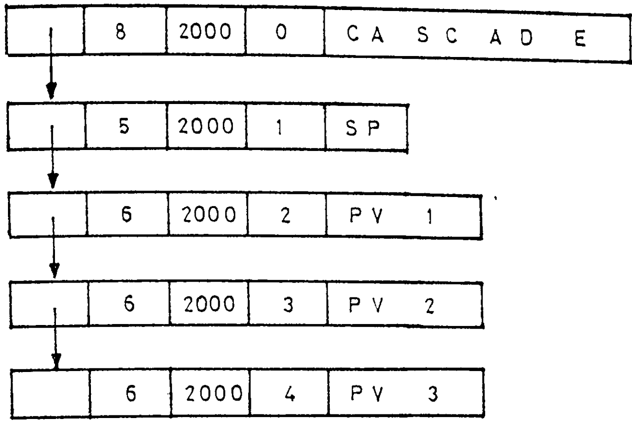
Block No. coords = (10,175)

I/O coords = (-10,140), (-10,110), (-10,80), (-10,50), (110,95)

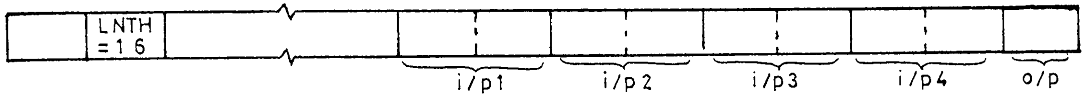
Picture coords = (-10+2000,25), (-10,165), (110,165), (110,25),

(-10,25)

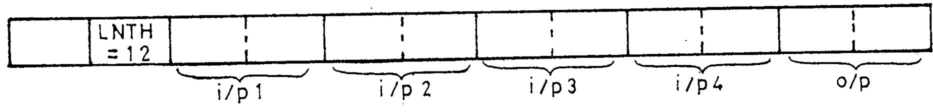
Type-specific Text Nodes:



Block Node:

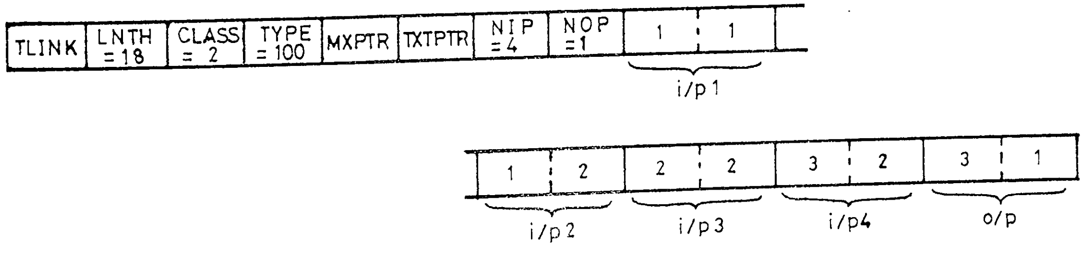


Macro Expansion Node:



Run-time Data Structure:

Composite Block Type Node:



A.3 Data Structures

The more important components of the Graphic and Run-time Data Structures are shown in Fig A.3 and Fig A.4 .

A.4 Main Software Components

The data module used in the run-time system is shown in Fig A.5 . All variables are declared to be externally accessible by other modules.

Figures A.6 to A.11 are source listings of the procedures for each function. The external data declarations (similar to the ones in the data module) are not shown as they will be inserted by the GPCL compiler. The three interface blocks use procedures which involve machine dependent code; this code is shown after the CORAL 66 procedure, although it is physically separated from it in actuality.

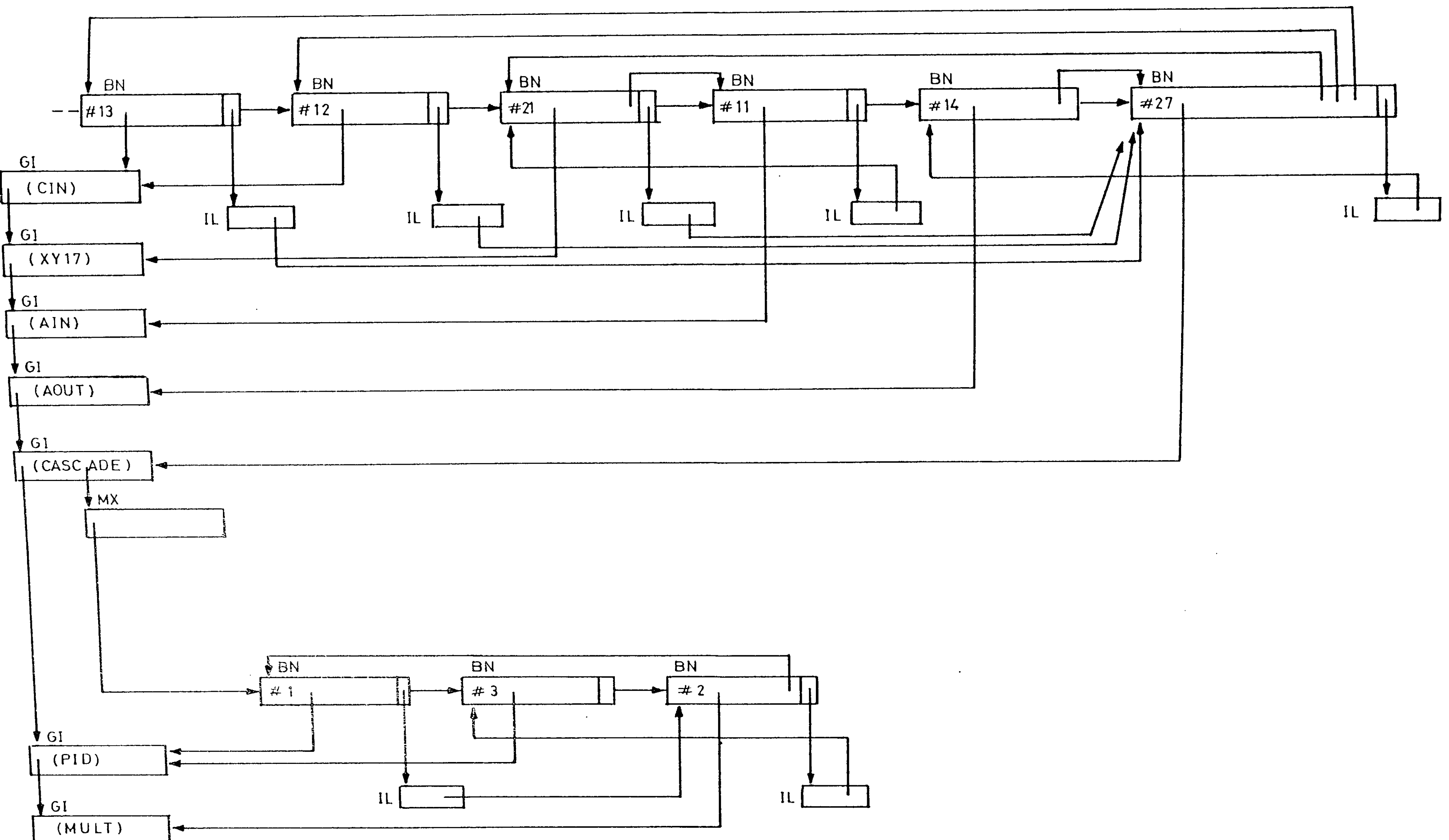


Figure A.3 Graphical Data Structure Example (Part)

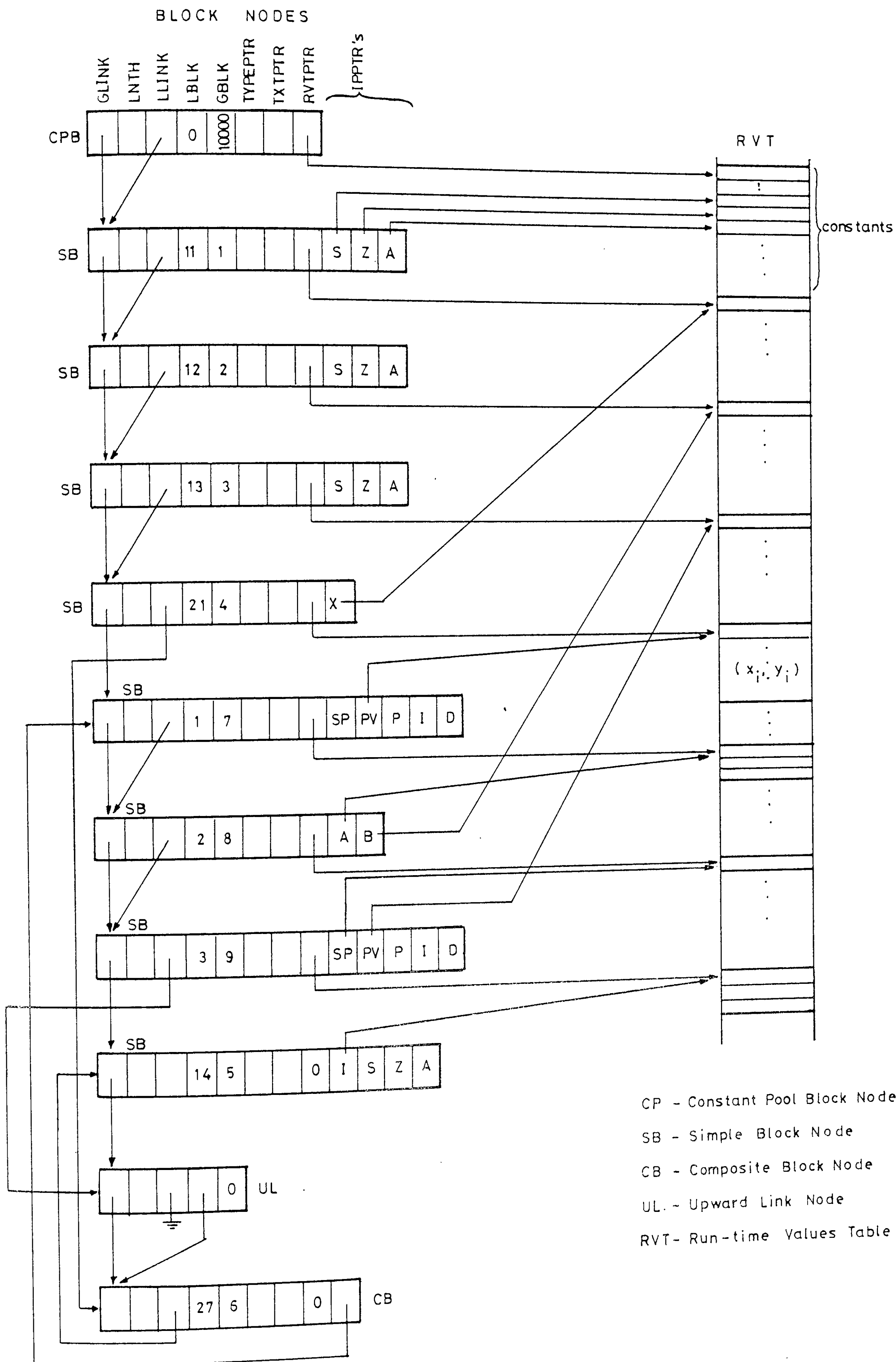


Figure A.4 Run-time Data Structure Example (Part)

```
'EXTERNAL' ('INTEGER' BLKNODE ; (PTR TO BLOCK NODE)
            'INTEGER' TYPNODE ; (PTR TO TYPE NODE)
            'INTEGER' OPBASE ; (PTR TO RVT ENTRIES)
            'INTEGER' IPBASE ; (PTR TO 1ST I/P PTR)
            'INTEGER' INITIAL ; (1ST ROUND FLAG)
            'INTEGER' INTERVAL ; (PROCESSING INTERVAL)
            'INTEGER' TIMEH, TIMEL ; (ABS. TIME IN MSEC)
            'INTEGER' HOUR, MINUTE, SECOND, MSECOND ;
            'INTEGER' CI1, CI2, CI3, CI4, CI5 ; (GENERAL)
            'FLOATING' CF1, CF2, CF3, CF4, CF5 ; (GENERAL)
            'INTEGER''ARRAY' RDS[ 0:1000 ] ;
            'FLOATING''ARRAY' RVT[ 0:200 ] ) ;

'SEGMENT' ONE

'BEGIN'

        'INTEGER' BLKNODE, TYPNODE, OPBASE, IPBASE,
                  INITIAL, INTERVAL, TIMEH, TIMEL,
                  HOUR, MINUTE, SECOND, MSECOND ;
        'INTEGER' CI1, CI2, CI3, CI4, CI5 ;
        'FLOATING' CF1, CF2, CF3, CF4, CF5 ;
        'INTEGER''ARRAY' RDS[ 0:1000 ] ;
        'FLOATING''ARRAY' RVT[ 0:200 ] ;

'END'
```

Figure A.5 Data Module

```
'EXTERNAL' ('PROCEDURE' PID);

'PROCEDURE' PID;

'DEFINE' V "RVT[ OPBASE ]";
'DEFINE' INT "RVT[ OPBASE+1 ]";
'DEFINE' OLDERR  "RVT[ OPBASE+2 ]";
'DEFINE' SP "RVT[ RDS[ IPBASE ] ]" ;
'DEFINE' PV "RVT[ RDS[ IPBASE+1 ] ]" ;
'DEFINE' KP "RVT[ RDS[ IPBASE+2 ] ]" ;
'DEFINE' KI "RVT[ RDS[ IPBASE+3 ] ]" ;
'DEFINE' KD "RVT[ RDS[ IPBASE+4 ] ]" ;

'BEGIN'
        'FLOATING' ERR;

        ERR:= SP-PV ;
        INT:= INTERVAL*ERR + INT ;
        V:= KP*ERR + KI*INT + KD*(ERR-OLDERR)/INTERVAL ;
        OLDERR:= ERR
'END';

'DELETE' KD; 'DELETE' KI; 'DELETE' KP;
'DELETE' PV; 'DELETE' SP; 'DELETE' OLDERR;
'DELETE' INT; 'DELETE' V;
```

Figure A.6 P-I-D Routine

```
'EXTERNAL'   ('PROCEDURE' AINPUT ;
              'PROCEDURE' INADC  ) ;



'PROCEDURE' AINPUT ;

'DEFINE' VALUE    "RVT[ OPBASE ]" ;
'DEFINE' SCALE    "RVT[ RDS[ IPBASE ] ]" ;
'DEFINE' ZERO     "RVT[ RDS[ IPBASE+1 ] ]" ;
'DEFINE' ADDRESS  "RVT[ RDS[ IPBASE+2 ] ]" ;

'BEGIN'
        CI1:= ADDRESS ;
        INADC ; (CALL ADC INPUT ROUTINE)
        VALUE:= CI2*SCALE + ZERO
'END';

'DELETE' ADDRESS; 'DELETE' ZERO; 'DELETE' SCALE; 'DELETE' VALUE;
```

```
*     ADC INPUT ROUTINE
        IDT   'INADC'
        DEF   INADC
        REF   CI1,CI2          EXTERNAL DATA
        REF   ADCADR           ADC ADDRESS
CRUBAS  EQU   12
*
INADC   MOV   @CI1,CRUBAS      /ADDRESS OF
        SLA   CRUBAS,1         / INPUT LINE
        SBO   0                ENABLE INPUT
        MOV   CRUBAS,R1        SAVE ADDRESS
        LI    CRUBAS,ADCADR         LOAD ADC ADDRESS
        SBZ   0                /STROBE TO
        SBO   0                / INITIATE
        SBZ   0                /  CONVERSION
LOOP    TB    14               A-D COMPLETE?
        JNE   LOOP             LOOP IF NOT
        STCR  R0,12            READ ADC
        MOV   R1,CRUBAS        /DISABLE
        SBZ   0                / INPUT
        AI    R0,-2048         CONVERT INTO 2'S COMPLEMENT
        MOV   R0,@CI2          PLACE ANSWER IN CI2
        RT                     RETURN
        END
```

Figure A.7 Analog Input Routines

```
'EXTERNAL'   ('PROCEDURE' CINPUT ;
              'PROCEDURE' INCTR ;
              'PROCEDURE' RSTCTR ) ;



'PROCEDURE' CINPUT ;

'DEFINE' CNT       "RVT[ OPBASE ]" ;
'DEFINE' SCALE     "RVT[ RDS[ IPBASE ] ]" ;
'DEFINE' ZERO      "RVT[ RDS[ IPBASE+1 ] ]" ;
'DEFINE' ADDRESS   "RVT[ RDS[ IPBASE+2 ] ]" ;

'BEGIN'
        CI1:= ADDRESS ;
       'IF' INITIAL = 0 'THEN' INCTR 'ELSE' RSTCTR ;
        (RESET COUNTER IF FIRST ROUND, ELSE READ)
        CNT:= CI2*SCALE + ZERO
'END';

'DELETE' ADDRESS; 'DELETE' ZERO; 'DELETE' SCALE; 'DELETE' CNT;
```

```
*  COUNTER INPUT AND RESET ROUTINES
        IDT   'INCTR'
        DEF   INCTR,RSTCTR
        REF   CI1,CI2
CRUBAS  EQU   12
*
INCTR   MOV   @CI1,CRUBAS    /LOAD COUNTER
        SLA   CRUBAS,1       / ADDRESS
        STCR  @CI2,0         READ COUNTER VALUE INTO CI2
* RESET COUNTER
RSTCTR  MOV   @CI1,CRUBAS    /LOAD COUNTER
        SLA   CRUBAS,1       / ADDRESS
        SBZ   0                   /RESET
        SBO   0                   / STROBE
        RT                   RETURN
        END
```

Figure A.8 Counter Input Routines

```
'EXTERNAL'   ('PROCEDURE' ANOUT  ;
              'PROCEDURE' OUTADC ) ;



'PROCEDURE' ANOUT ;

'DEFINE' VALUE    "RVT[ RDS[ IPBASE ] ]" ;
'DEFINE' SCALE    "RVT[ RDS[ IPBASE+1 ] ]" ;
'DEFINE' ZERO     "RVT[ RDS[ IPBASE+2 ] ]" ;
'DEFINE' ADDRESS  "RVT[ RDS[ IPBASE+3 ] ]" ;

'BEGIN'
       'FLOATING' V;
        V:= (VALUE-ZERO)/SCALE ;
       'COMMENT' V IS EXPECTED TO BE POSITIVE VALUE
                 WITHIN 0 TO 4095 ;
       'IF' V<0 'THEN' 'BEGIN'
                              CI2:=0 ; 'GOTO' OUTPUT
                        'END';
        CI2:= 'IF' V<4096 'THEN' V 'ELSE' 4095 ;
OUTPUT:
        CI1:= ADDRESS ;
        OUTADC
'END';

'DELETE' ADDRESS; 'DELETE' ZERO; 'DELETE' SCALE; 'DELETE' VALUE;
```

```
*  DAC OUTPUT ROUTINE
       IDT  'OUTDAC'
       DEF  OUTDAC
       REF  CI1,CI2         EXTERNAL DATA
       REF  DACADR          DAC ADDRESS
CRUBAS EQU  12
*
OUTDAC LI   CRUBAS,DACADR        LOAD DAC ADDRESS
       LDCR @CI2,12         OUTPUT VALUE TO DAC
       MOV  @CI1,CRUBAS     /LOAD OUTPUT
       SLA  CRUBAS,1        / ADDRESS
       SBZ  0                    /SAMPLE
       SBO  0                    / AND
       SBZ  0                    /  HOLD
       RT                   RETURN
       END
```

Figure A.9 Analog Output Routines

```
'EXTERNAL' ('PROCEDURE' XYFUNCTION);



'PROCEDURE' XYFUNCTION;

'DEFINE' OUTPUT "RVT[ OPBASE ]";
'DEFINE' X(N)    "RVT[ N*2+OPBASE-1 ]";
'DEFINE' Y(N)    "RVT[ N*2+OPBASE ]";
'DEFINE' XIN     "RVT[ RDS[ IPBASE] ]";
'DEFINE' NPOINTS "RVT[ RDS[ TYPEPTR+6 ] ]";

'BEGIN'
        'INTEGER' I,J,STEP,SIGN ;

         STEP:=NPOINTS/2 ;
         I:=1 ;
         SIGN:=1 ;

LOOP:    I:=I+STEP*SIGN ;
        'IF' XIN < X(I) 'THEN' SIGN:=-1 'ELSE' SIGN:=1 ;
         STEP:=STEP/2 ;
        'IF' STEP > 0 'THEN''GOTO' LOOP ;

'COMMENT' INTERPOLATE;
         J:=I+SIGN ;
         OUTPUT:= (XIN-X(I))*(Y(J)-Y(I))/(X(J)-X(I))
                      +Y(I)
'END';

'DELETE' NPOINTS; 'DELETE' XIN; 'DELETE' Y;
'DELETE' X;         'DELETE' OUTPUT;
```

Figure A.10 X-Y Function Routine

```
'EXTERNAL' ('PROCEDURE' MULTIPLIER);



'PROCEDURE' MULTIPLIER;
'DEFINE' C "RVT[ OPBASE ]";
'DEFINE' A "RVT[ RDS[ IPBASE ] ]";
'DEFINE' B "RVT[ RDS[ IPBASE+1 ] ]";
'BEGIN'
        C := A*B
'END';
'DELETE' B;  'DELETE' A;  'DELETE' C;
```

Figure A.11 Multiplier Routine

REFERENCES

1. PIKE H.E.: 'Process control software', Proc. IEEE, vol.58, no.1, January 1970, p.87-97.

2. SCHOEFFLER J.D. and TEMPLE R.H.: 'A real-time language for industrial process control', Proc. IEEE, vol.58, no.1, January 1970, p.98-111.

3. FRAADE D.J. and GAST S.: 'A survey of computer networks and distributed control', Proc. 5th IFAC/IFIP Int. Conf. on Digital Computer Applications to Process Control, The Hague, June 1977, van Naute Lemke H.R. and Verbruggen H.B. (Eds), Elsevier North-Holland Inc., NY, 1977, p.13-30.

4. REYNOLDS B.: 'The principles of total distributed control', Proc. Symposium on Dedicated Digital Control, Univ. of Aston in Birmingham, UK, April 1977.

5. DUYFYES G., DE JONG P.J. and VERBRUGGEN H.B.: 'Questionaire on applications of modern control theory to computer control in the process industry - results and comments', Proc. 5th IFAC/IFIP Int. Conf. on Digital Computer Applications to Process Control, The Hague, June 1977, van Naute Lemke H.R. and Verbruggen H.B. (Eds), Elsevier North-Holland Inc., NY, 1977, p.833-841.

6. BROWN T.J.: 'Modular approach to control', Proc. Symposium on Dedicated Digital Control, Univ. of Aston in Birmingham, UK, April 1977.

7. GALLACHER J. and THORNLEY G.A.: 'The design of a modular systems applications microcomputer', Proc. Symposium on Dedicated Digital Control, Univ. of Aston in Birmingham, UK, April 1977.

8. BASS R.J. and DONOVAN J.: 'The automatic control of sugar crystallisation', Proc. Symposium on Dedicated Digital Control, Univ. of Aston in Birmingham, UK, April 1977.

9. EMMERSON E.: 'Microprocessor application to a counter pressure autoclave', Proc. 3rd Int. Conf. on Trends in On-line Computer Control Systems, Sheffield, UK, March 1979, IEE, London, 1979, p.66-71.

10. WILKIE J.D.F.: 'A microprocessor philosophy for process control systems', Proc. 3rd Int. Conf. on Trends in On-line Computer Control Systems, Sheffield, UK, March 1979, IEE, London, 1979, p.115-120.

11. QUITTNER G.F.: 'Quo vadis automation?', IEEE Trans. Industrial Electronics and Control Instrumentation, vol.IECI-21, no.4, November 1974, p.215-221.

12. WEISS E.A.: 'Introduction: computer languages for process control', IEEE Trans. Industrial Electronics and Control Instrumentation, vol.IECI-16, no.3, December 1969, p.180.

13. OSTFIELD D.M.: 'Standardisation - the questions to ask', IEEE Trans. Industrial Electronics and Control Instrumentation, vol.IECI-16, no.3, December 1969, p.181-182.

14. HOHMEYER R.E.: 'Standardisation - the February 1969 Purdue Workshop', IEEE Trans. Industrial Electronics and Control Instrumentation, vol.IECI-16, no.3, December 1969, p.183-185.

15. HEPBURN G.A.: 'Direct digital control in CANDU nuclear power plants - maximum utilisation of a dual redundant minicomputer system', Proc. 3rd Int. Conf. on Trends in On-line Computer Control Systems, Sheffield, UK, March 1979, IEE, London, 1979, p.96-99.

16. THOMPSON K.: 'Survey report on real-time languages standards for industrial use', Minicomputer Forum, Conf. Proc., On-line Conferences Ltd, Uxbridge, Middx, UK, 1975, p.121-154.

17. ORGANICK E.I. and MEISSNER L.P.: FORTRAN IV, 2nd ed., Reading, Mass.; London etc: Addison-Wesley, 1974.

18. ICL 1900 Series: FORTRAN, 2nd ed., London : International Computers Ltd, 1971.

19. JARVIS P.H.: 'Some experiences with process control languages', IEEE Trans. Industrial Electronics and Control Instrumentation, vol.IECI-15, no.2, December 1968, p.54-56.

20. MENSH M. and DIEHL W.: 'Extended FORTRAN for process control', IEEE Trans. Industrial Electronics and Control Instrumentation, vol.IECI-15, no.2, December 1968, p.75-79.

21. HOHMEYER R.E.: 'CDC1700 FORTRAN for process control', IEEE Trans. Industrial Electronics and Control Instrumentation, vol.IECI-15, no.2, December 1968, p.67-70.

22. FROST D.R.: 'Computer languages for process control - their future', IEEE Trans. Industrial Electronics and Control Instrumentation, vol.IECI-16, no.3, December 1969, p.189-192.

23. PIKE H.E.: 'Future trends in software development for real-time industrial automation', Proc. 1972 Spring Joint Computer Conf., AFIPS Conf. Proc., p.915-923.

24. GRONER G.F.: PL/I programming in technological applications, New York; Chichester : Wiley-Interscience, 1971.

25. GEAR C.W.: PL/I and PL/C language manual, (Chigago) : Science Research Associates, 1978.

26. Algol - ICL student edition, International Computers Ltd, - London : International Computers Ltd, 1970.

27. WOODWARD P.M., WETHERALL P.R. and GORMAN B.: Official definition of CORAL 66, 3rd ed., HMSO, London, England, 1974.

28. HALLIWELL J.D. and EDWARDS T.A.: A course in standard CORAL 66, NCC Publications, National Computing Centre Ltd, Manchester, Endland, 1977.

29. BARNES J.G.P.: RTL/2 design and philosophy, Heyden and Son Ltd., London, UK, 1976.

30. BARNES J.G.P.: 'The use fo RTL/2 for multi-tasking', Minicomputer Forum, Conf. Proc., On-line Conferences Ltd, Uxbridge, Middx, UK, 1975, p.157-166.

31. FREVERT L.: 'Realtime language PEARL - concepts of the language design and implementation', Minicomputer Forum, Conf. Proc., On-line Conferences Ltd, Uxbridge, Middx, UK, 1975, p.183-191.

32. ELZER P. and ROESSLER R.: 'Realtime languages and operating systems', Proc. 5th IFAC/IFIP Int. Conf. on Digital Computer Applications to Process Control, The Hague, June 1977, van Naute Lemke H.R. and Verbruggen H.B. (Eds), Elsevier North-Holland Inc., NY, 1977, p.1-12.

33. GASPER T.G. and DOBROHOTOFF V.V.: 'New process language uses English terms', Control Engineering, October 1968, p.118-121.

34. JACKSON S.P.: 'Current trends in automation - state of the art in computer control of operations', IEEE Trans. Industrial Electronics and Control Instrumentation, August 1973, p.107-114.

35. JACKSON S.P. and BIESECKER F.S.: 'General purpose automatic control system for the chemical industry', Instrumentation in the Chemical and Petroleum Industries, vol.9, St.Louis, MO, USA, April 1973, p.45-48.

36. HERTANU H.I.: 'High-level languages for programming DDC and/or sequential control in process plants', Advances in Instrumentation, vol.29, Part IV, New York, USA, October 1974, (Pittsburgh, Pa, USA : ISA 74), p.817/1-817/4.

37. NOBLE J.S.: 'The evolution of process control software', Proc. Symposium on Dedicated Digital Control, Univ. of Aston in Birmingham, UK, April 1977.

38. BATES D.G.: 'PROSPRO/1800', IEEE Trans. Industrial Electronics and Control Instrumentation, vol.IECI-15, no.2, December 1968, p.70-75.

39. --, Bristol UCS 3000 'The Process Controller', (Technical Bulletin TM280A), American Chain & Cable Co. Inc., USA, 1975.

40. SPURLING K.: 'The organisation and support of colour displays for a nuclear research accelerator', Proc. 3rd Int. Conf. on Trends in On-line Computer Control Systems, Sheffield, UK, March 1979, IEE, London, 1979, p.41-44.

41. KOMPASS E.J.: 'Control users speak out on microprocessors', Control Engrng, December 1975, p.35-37.

42. HARRIS L. and BELL T.: 'Symbolic function-oriented programming for process control has simple rules', Control Engrng, February 1976, p.91-92.

43. KOSSIAKOFF A. and SLEIGHT T.P.: 'A programming language for real-time systems', Proc. 1972 Fall Joint Computer Conf., AFIPS Conf. Proc., p.923-942a.

44. GORDON-CLARK M.R.: Session 1 General Discussion, IFAC-IFIP Workshop on Real-time Programming, Boston, August 1975, p.26.

45. BROWN P. and WALKER R.W.: 'COMPASS - an engineer oriented computer on-line multipurpose application software system', Proc. 3rd Int. Conf. on Trends in On-line Computer Control Systems, Sheffield, UK, March 1979, IEE, London, 1979, p.49-52.

46. CROWLEY-MILLING M.C.: 'The data module, the missing link in high level control languages', Proc. 3rd Int. Conf. on Trends in On-line Computer Control Systems, Sheffield, UK, March 1979, IEE, London, 1979, p.63-65.

47. DIEHL W.: 'Software for industrial computer control - a review', Proc. 1st IFAC/IFIP Symposium on Software for Computer Control, Sococo 1976, p.279-285.

48. SCHOEFFLER J.D.: 'Some views on standardisation', IEEE Trans. Industrial Electronics and Control Instrumentation, vol.IECI-16, no.3, December 1969, p.185-187.

49. ALGOL 68: users guide, Univ. Manchester Regional Computer Centre - Manchester : UMRCC, 1979.

50. ALGOL 68-R system: programmer's manual', Malvern : Royal radar Establishment, Division of Computing and Software Research, 1971.

51. SCHOEFFLER J.D.: 'The development of process control software', Proc. 1972 Spring Joint Computer Conf., AFIPS Conf. Proc., p.907-914.

52. CLARKE D.W. and FROST P.J.: 'A control BASIC for microcomputers', Proc. 3rd Int. Conf. on Trends in On-line Computer Control Systems, Sheffield, UK, March 1979, IEE, London, 1979, p.53-57.

53. HEHER A.D.: 'Some features of a real-time BASIC executive', Software - Practice and Experience, vol.6, July-September, 1976, p.387-391.

54. BISHOP P.G., BREWER C. and JERVIS P.: 'The design of software for distributed computer control systems', Proc. 3rd Int. Conf. on Trends in On-line Computer Control Systems, Sheffield, UK, March 1979, IEE, London, 1979, p.58-62.

55. EELDERINK G.H.B., BRUIJIN P.M. and VERBRUGGEN H.B.: 'Microprocessor based controllers in a distributed network implemented with a block-diagram language', Proc. 3rd Int. Conf. on Trends in On-line Computer Control Systems, Sheffield, UK, March 1979, IEE, London, 1979, p.174-176.

56. BOS J.F., JANSEN J.P. and MAARLEVELD A.: 'ICOSS: an integrated approach to process automation', Proc. 5th IFAC/IFIP Int. Conf. on Digital Computer Applications to Process Control, The Hague, June 1977, van Naute Lemke H.R. and Verbruggen H.B. (Eds), Elsevier North-Holland Inc., NY, 1977, p.637-644.

57. DEBELLE J., DECOUSSEMAEKER J., NUYTS R., VASQUEZ R., MARQUE P. and VAN DEMOOSDYK R.: 'First Belgian application of a digital computer for the control of a 280MW boiler of the terminal power station at Genk-Langerlo', Proc. 5th IFAC/IFIP Int. Conf. on Digital Computer Applications to Process Control, The Hague, June 1977, van Naute Lemke H.R. and Verbruggen H.B. (Eds), Elsevier North-Holland Inc., NY, 1977, p.769-787.

58. GRONER G.F., CLARK R.L., BERMAN R.A. and DELAND E.C.: 'BIOMOD- an interactive graphics system for modelling', Proc. 1971 Fall Joint Computer Conf., AFIPS Conf. Proc., p.369-378.

59. GOLDSTEIN R.A. and NAGEL R.: '3-D visual simulation', Simulation, vol.16, no.1, 1972, p.25.

60. RAHI G.A.: 'Simulation and computer graphics', Simulation, vol.19, no.2, August 1972, p.13-16.

61. WEINBERG R.: 'Computer graphics in support of space shuttle simulation', Proc. SIGGRAPH '78, 5th Annual Conference on Computer Graphics and Interactive Techniques, Atlanta, Ga, August 1978, ACM, NY, 1978, p.354-356.

62. BRACCHI G. and SOMALVICO M.: 'An interactive software system for computer-aided design : an application to circuit project', CACM, vol.13, no.9, September 1970, p.537-545.

63. DAVY D.W. and BRAND D.E.: 'Computer graphic aids to car body design', Computer Graphics 70, Int. Symposium, Brunel Univ., Uxbridge, Middx, UK, vol.3 .

64. FRANKLIN J.L. and DEAN E.B.: 'Interactive graphics for computer aided network design', AFIPS Conf. Proc., 1973 Nat. Computer Conf. Expo., vol.42, 1973, p.677-683.

65. ARGYRIS J.H. and GRIEGER I.: 'Interactive computer graphics in structural analysis', Proc. On-line 72, Int. Conf. on On-line Interactive Computing, Uxbridge, Middx, UK, September 1972, On-line Computer Systems Ltd, 1972, p.709-725.

66. CHACE M.A. and KORYBALSKI M.E.: 'Computer graphics in the dynamic analysis of mechanical networks', Computer Graphics 70, Int. Symposium, Brunel Univ., Uxbridge, Middx, UK, vol.3, p.115-160.

67. TALBOT P.A., CARR J.W., COULTER R.R. and HWANG R.C.: 'Animator : an on-line two-dimensional film animation system', CACM, vol.14, no.4, April 1971, p.251-259.

68. HERBISON-EVANS D.: 'NUDES2: a numeric utility displaying ellipsoid solids, version 2', Proc. SIGGRAPH '78, 5th Annual Conference on Computer Graphics and Interactive Techniques, Atlanta, Ga, August 1978, ACM, NY, 1978, p.354-356.

69. NEWMAN W.M. and SPROULL R.F.: 'An approach to graphics system design', Proc. IEEE, vol.62, no.4, April 1974, p.471-483.

70. GAMMILL R.C. and ROBERTSON D: 'Graphics and interactive systems - design cosiderations of a software system', Proc. 1973 Nat. Computer Conf., AFIPS Conf. Proc., p.657-662.

71. RAPKIN M.D. and ABU-GHEIDA O.M.: 'Stand-alone/remote graphic systems', Proc. 1968 Fall Joint Computer Conference, Washington D.C. : Thompson, 1968, p.731-746.

72. WOODSFORD P.A.: 'The design and implementation of the GINO 3D graphics software package', Software Practice and Experience, vol.1, October 1971, p.335-365.

73. HEILMAN R.L. and MARCHANT J.M.: 'TIGS - an overview of the Terminal Independent Graphics System', Proc. SIGGRAPH '78, 5th Annual Conference on Computer Graphics and Interactive Techniques, Atlanta, Ga, August 1978, ACM, NY, 1978, p.293-297.

74. WARNER J.R., POLISHER M.A. and KOPOLOW R.N.: 'DIGRAF - a FORTRAN implementation of the proposed GSPC standard', Proc. SIGGRAPH '78, 5th Annual Conference on Computer Graphics and Interactive Techniques, Atlanta, Ga, August 1978, ACM, NY, 1978, p.301-307.

75. KELLNER R.G., REED T.N. and SOLEM A.V.: 'An implementation of the ACM/SIGGRAPH proposed graphics standard in a multisystem environment', Proc. SIGGRAPH '78, 5th Annual Conference on Computer Graphics and Interactive Techniques, Atlanta, Ga, August 1978, ACM, NY, 1978, p.308-312.

76. BERGERON R.D., BONO P.R. and FOLEY J.D.: 'Graphics programming using the Core system', Computing Surveys, vol.10, no.4, December 1978, p.389-442.

77. KULSRUD H.E.: 'A general-purpose graphic language', CACM, vol.11, no.4, April 1968, p.247-254.

78. NEWMAN W.M.: 'An informal graphics system based on the LOGO language', Proc. 1973 Nat. Computer Conf., AFIPS Conf. Proc., p.651-655.

79. ABRAMS M.D.: 'Data structures for computer graphics', Proc. A Symposium on Data Structures in Programming Languages, SIGPLAN Notices, vol.6, no.2, February 1971, p.268-286.

80. SUTHERLAND I.E.: 'SKETCHPAD - a man-machine graphical communication system', Proc. 1963 Spring Joint Computer Conf., AFIPS Conf. Proc., p.329-345.

81. COTTON I.W. and GREATOREX F.S.: 'Data structures and techniques for remote computer graphics', Proc. 1968 Fall Joint Computer Conf. AFIPS Conf. Proc., p.533-544.

82. VAN DAM A.: 'Data and storage structures for interactive graphics' Proc. A Symposium on Data Structures in Programming Languages, SIGPLAN Notices, vol.6, no.2, February 1971, p.237-267.

83. CARDENAS A.F. and SEELEY R.W.: 'A simple data structure for interactive graphics design/drafting', The Computer Journal, vol.18, no.1, 1975, p.30-32.

84. VAN DAM A. and EVANS D.: 'A compact data structure for storing, retrieving and manipulating line drawings', Proc. 1967 Spring Joint Computer Conf., AFIPS Conf. Proc., p.601-610.

85. SHAPIRO L.G.: 'Data structures for picture processing', Proc. SIGGRAPH '78, 5th Annual Conference on Computer Graphics and Interactive Techniques, Atlanta, Ga, August 1978, ACM, NY, 1978, p.140-146.

86. DAVIS M.R. and ELLIS T.O.: 'The RAND tablet: a man machiine communication device', Proc. 1964 Fall Joint Computer Conf., AFIPS Conf. Proc., p.325-331.

87. NEWMAN W.M. and SPROULL R.F.: Priciples of Interactive Computer Graphics, New York: McGraw-Hill, 1973.

88. COTTON I.W.: 'Network graphic attention handling', Proc. On-line 72, Int. Conf. on On-line Interactive Computing, Uxbridge, Middx, UK, September 1972, On-line Computer Systems Ltd, 1972, p.465-490.

89. FOLEY J.D. and WALLACE V.L.: 'The art of natural graphic man-machine conversation', Proc. IEEE, vol.62, no.4, April 1974, p.462-470.

90. NEWMAN W.M.: 'A system for interactive graphical programming', Proc. 1968 Spring Joint Computer Conf., AFIPS Conf. Proc., p.47-54.

91. OHLSON M.: 'System design considerations for graphics input devices', Computer, November 1978, p.9-18.

92. CARLSON E.D.: 'Graphics terminal requirements for the 1970's ', Computer, August 1976, p.37-45.

93. LUCIDO A.P.: 'An overview of directed beam graphics display hardware', Computer, November 1978, p.29-36.

94. PREISS R.B.: 'Storage CRT display terminals: evolution and trends', Computer, November 1978, p.20-26.

95. NEWMAN W.M. and VAN DAM A.: 'Recent efforts towards graphics standardisation', Computing Surveys, vol.10, no.4, December 1978, p.365-380.

96. MICHENER J.C. and VAN DAM A.: 'A functional overview of the Core system with glossary', Computing Surveys, vol.10, no.4, December 1978, p.381-387.

97. CHRISTENSON C. and PINSON E.N.: 'Multi-function graphics for a large computer system', Proc. 1967 Fall Joint Computer Conf., AFIPS Conf. Proc., p.697-711.

98. FRANK A.J.: 'B-LINE, Bell line drawing language', Proc. 1968 Fall Joint Computer Conf., AFIPS Conf. Proc., p.179-191.

99. BERHOLD E.D., BERHOLD M.P. and MCNAMEE L.P.: 'Structural operational data sets from arbitrarily arranged computer graphic symbols', Computer Graphics 70, Int. Symposium, Brunel Univ., Uxbridge, Middx, UK, vol.3, p.161-178.

100. SMITH G.W. and WOOD R.C.: in Principles of Analog Computation, New York: McGraw-Hill 1959, p.113-115.

101. JOHNSON C.L.: in Analog Computer Techniques, 2nd ed., New York: McGraw-Hill 1963, p.118-119.

102. HEALEY M.: 'A survey of minicomputer applications in industrial control', Minicomputer Forum, Conf. Proc., On-line Conferences Ltd, Uxbridge, Middx, UK, 1975, p.493-503.

103. STEPHENSON R.E.: Computer Simulation for Engineers, New York: Harcourt Brace Jovanich, 1971.

104. CHU YAOHAN: Digital Simulation of Continuous Systems, New York: McGraw-hill 1969.

105. SCi SIMULATION SOFTWARE COMMITTEE: 'The SCi Continuous System Simulation Language (CSSL)', Simulation, December 1967, p.281-303.