

# THE DESCRIPTOR



*a definition of the B5000 information processing system*

BULLETIN 5000-20002-P  
FEBRUARY 1961

**THE DESCRIPTOR**  
– a definition of the B 5000 Information Processing System

Sales Technical Services  
Equipment and Systems Marketing Division

**Burroughs Corporation**  
Detroit 32, Michigan

Copyright © 1961 Burroughs Corporation

## TABLE OF CONTENTS

Section	Page
1	INTRODUCTION . . . . . 1-1
2	PROGRAMMING LANGUAGE . . . . . 2-1
	ADVANTAGES OF PROBLEM-ORIENTED LANGUAGES . . . . . 2-1
	Computational Language (ALGOL) . . . . . 2-1
	Data-Processing Language (COBOL) . . . . . 2-1
	Reduced Compiling and Running Time . . . . . 2-2
	Simplified Debugging and Program Maintenance . . . . . 2-2
	PROGRAMMING ADVANTAGES OF THE B 5000 SYSTEM . . . . . 2-3
	Reduced Programming Time . . . . . 2-3
	Reduced Compiling and Running Time . . . . . 2-3
	Conventional Compiler Functions . . . . . 2-3
	Burroughs Compiler Techniques . . . . . 2-4
	Simplified Debugging and Program Maintenance . . . . . 2-8
3	SYSTEM CHARACTERISTICS . . . . . 3-1
	GENERAL DESCRIPTION . . . . . 3-1
	Storage Drum . . . . . 3-1
	Core Memory . . . . . 3-1
	Words . . . . . 3-2
	Program Reference Table . . . . . 3-3
	Program Segments . . . . . 3-3
	The Stack . . . . . 3-3
	Input/Output Areas . . . . . 3-5
	Memory Addresses . . . . . 3-5
	Processing . . . . . 3-5
	Modes of Operation . . . . . 3-5
	OPERATION DURING ARITHMETIC MODE . . . . . 3-5
	Introduction . . . . . 3-5
	Program Reference Table . . . . . 3-7
	Descriptors . . . . . 3-8
	Operands . . . . . 3-9
	Operation of Program Syllables in Arithmetic Mode . . . . . 3-9
	Summary . . . . . 3-15
	OPERATIONS DURING SUBROUTINE MODE . . . . . 3-15
	Introduction . . . . . 3-15
	Use of Operand-Call and Descriptor-Call Syllables . . . . . 3-15
	The F Register . . . . . 3-15
	Special Subroutine Operators . . . . . 3-16
	Entry to Subroutine Mode . . . . . 3-17
	Exit from Subroutine Mode . . . . . 3-20
	OPERATION DURING DATA-MANIPULATION MODE . . . . . 3-21
	Introduction . . . . . 3-21
	Editing Functions . . . . . 3-22
	Entry to Data-Manipulation Mode . . . . . 3-22
	Illustration of Some Specific Operators Utilized by the Compilers . . . . . 3-23
	Data-Manipulation Mode Example 1 . . . . . 3-26
	Data-Manipulation Mode Example 2 . . . . . 3-27
	Data-Manipulation Mode Example 3 . . . . . 3-28

## TABLE OF CONTENTS (continued)

Section	Page
4	<b>MASTER CONTROL PROGRAM</b> ..... 4-1 <b>INTRODUCTION</b> ..... 4-1 Computer Functions ..... 4-1 Automatic System Assignment and Coordination ..... 4-1 Multi-Processing ..... 4-1 Elements of the Master Control Program ..... 4-1 Entry to the Control Mode ..... 4-2 The Interrupt Concept ..... 4-2 <b>EXECUTIVE ROUTINE</b> ..... 4-2 Input/Output Operations ..... 4-2 Initiation of an Input/Output Operation ..... 4-3 Use of the Continuity Bit ..... 4-3 Completion of an Input/Output Operation ..... 4-3 Summary ..... 4-4 Handling of Interrupt Conditions ..... 4-4 Processor-Dependent Interrupts ..... 4-4 Processor-Independent Interrupts ..... 4-5 Program Reject Routine ..... 4-5 Control of Program Segments ..... 4-5 Use of Other Master Control Program Routines ..... 4-6 Maintenance of an Operations Log ..... 4-6 Maintenance of System Description ..... 4-6 Summary ..... 4-6 <b>THE SCHEDULE ROUTINE</b> ..... 4-6 Program Backlog Table ..... 4-6 Input/Output Requirements ..... 4-7 Memory Requirements ..... 4-7 Summary ..... 4-7 <b>ENVIRONMENT CONTROL ROUTINE</b> ..... 4-7 Assignment of Input/Output Units ..... 4-7 Allocation of Memory ..... 4-7 Base Location Tables ..... 4-8 Loading Program Segments ..... 4-8 Loading Additional Segments ..... 4-8 Program-Finish Conditions ..... 4-8 Changing the Schedule ..... 4-9
5	<b>COMPONENTS</b> ..... 5-1 <b>INTRODUCTION</b> ..... 5-1 <b>CONSOLE</b> ..... 5-1 <b>MEMORY MODULE</b> ..... 5-1 <b>MEMORY EXCHANGE</b> ..... 5-3 <b>INPUT/OUTPUT EXCHANGE</b> ..... 5-3 <b>INPUT/OUTPUT SYSTEM</b> ..... 5-3 Input/Output Channel ..... 5-4 Input Information Flow ..... 5-4 Output Information Flow ..... 5-4 <b>STORAGE DRUM</b> ..... 5-7 <b>MAGNETIC TAPE UNIT</b> ..... 5-7 Read Operations ..... 5-8

## TABLE OF CONTENTS (continued)

Section	Page
5 (cont.) Write Operations	5-8
LINE PRINTER	5-9
CARD HANDLING EQUIPMENT	5-10
Card Readers	5-11
Card Punch	5-12
PLOTTER	5-12
MESSAGE PRINTER/KEYBOARD	5-13
Message Printer	5-13
Keyboard	5-13

## APPENDIXES

Appendix	Page	
A	ALGOL CHARACTERISTICS	A-1
	INTRODUCTION	A-1
	BASIC SYMBOLS AND WORDS	A-1
	Letters	A-1
	Digits	A-1
	Operators and Symbols	A-1
	Identifiers	A-1
	Variable	A-2
	Procedure	A-2
	Switch	A-2
	Array	A-2
	Label	A-2
	Formal Parameter	A-2
	Numbers	A-2
	Strings	A-2
	EXPRESSIONS	A-2
	Variables	A-2
	Functional Designators	A-2
	Arithmetic Expressions	A-3
	Simple	A-3
	General	A-3
	Boolean Expressions	A-3
	Simple	A-3
	General	A-3
	Designational Expressions	A-4
	STATEMENTS	A-4
	Assignment Statements	A-4
	Go To Statements	A-4
	Dummy Statements	A-4
	Conditional Statements	A-5
	For Statements	A-5
	Arithmetic Expression Element	A-5
	Step-Until Elements	A-5
	While Element	A-5
	Procedure Statements	A-5

## APPENDIXES (continued)

Appendix	Page
A (cont.)	DECLARATIONS . . . . . A-5
	Type Declarations . . . . . A-5
	Array Declarations . . . . . A-6
	Switch Declarations . . . . . A-6
	Procedure Declarations . . . . . A-6
	Procedure Declaration Heading . . . . . A-6
	Procedure Declaration Body . . . . . A-6
	ALGOL EXAMPLE . . . . . A-6
B	B 5000 DATA-PROCESSING LANGUAGE . . . . . B-1
	DEVELOPMENT . . . . . B-1
	FEATURES . . . . . B-1
	ADVANTAGES . . . . . B-1
	GENERAL DESCRIPTION . . . . . B-1
	IDENTIFICATION DIVISION . . . . . B-2
	ENVIRONMENT DIVISION . . . . . B-2
	DATA DIVISION . . . . . B-2
	PROCEDURE DIVISION . . . . . B-3
	DETAILED DESCRIPTION . . . . . B-3
	Character Set . . . . . B-3
	Characters Used in Forming Words . . . . . B-3
	Characters Used for Punctuation . . . . . B-4
	Characters Used in Relations . . . . . B-4
	Characters Used in Editing . . . . . B-5
	Words . . . . . B-5
	Nouns . . . . . B-5
	Qualification . . . . . B-5
	Subscripts . . . . . B-6
	Notation . . . . . B-6
	Verbs . . . . . B-6
	Procedures . . . . . B-8
	Conditionals . . . . . B-8
	Simple Condition . . . . . B-8
	Relations . . . . . B-8
	Compound Condition . . . . . B-9
	Statements . . . . . B-9
	Imperative Statements . . . . . B-9
	Conditional Statements . . . . . B-9
	Sentences . . . . . B-10
	Imperative Sentences . . . . . B-10
	Conditional Sentences . . . . . B-10
	Rules for Converting Flow Charts into Narrative Form . . . . . B-11
	Rules for Omitting Names . . . . . B-11
	Punctuation . . . . . B-12
	Sentence Execution . . . . . B-12
	Imperative Sentences . . . . . B-12
	Conditional Sentences . . . . . B-12
	Compiler Directing Sentences . . . . . B-12

## APPENDIXES (continued)

<b>Appendix</b>	<b>Page</b>
B (cont.)	
Control Relationship Between Sentences .....	B-12
Paragraph .....	B-13
Section .....	B-13
ACKNOWLEDGMENT .....	B-13
C GLOSSARY .....	C-1

## LIST OF ILLUSTRATIONS FIGURES

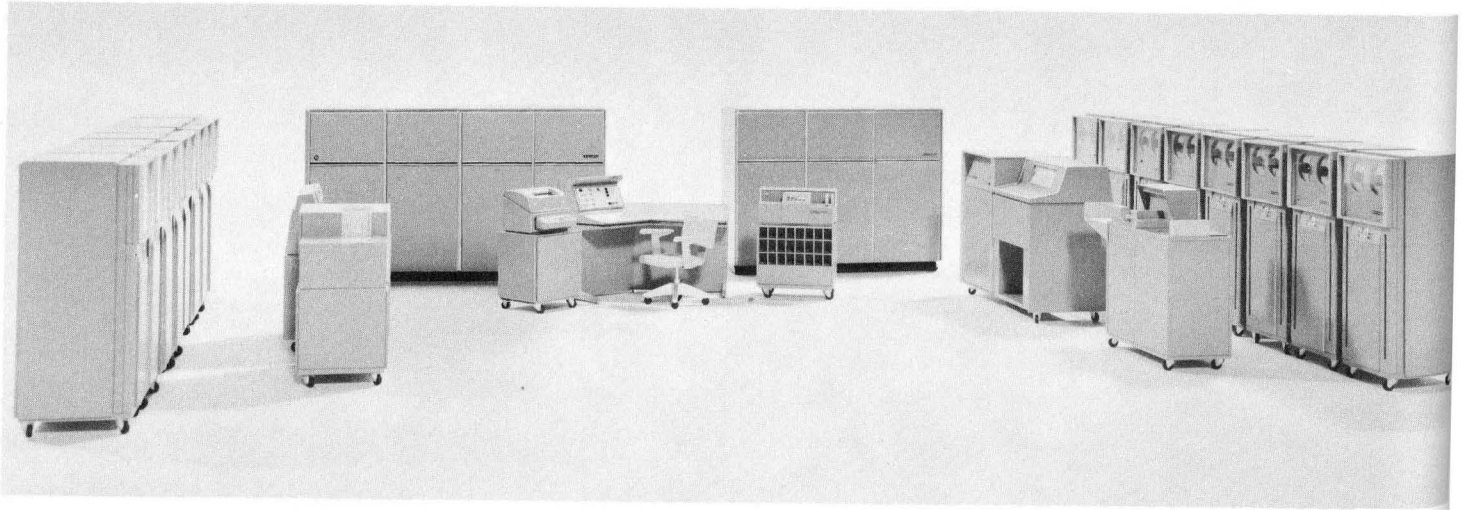
<b>Figure</b>	<b>Page</b>	
2-1	Flow Chart of Translation Process .....	2-6
2-2	Stages of the Scanning Process .....	2-7
3-1	Compilation of Source Program .....	3-1
3-2	Layout of the Drum .....	3-2
3-3	Allocation of Core Memory .....	3-2
3-4	Shifting of Information in the Stack .....	3-4
3-5	Condition of Stack and Associated Registers After Information Has Been Removed .....	3-4
3-6	Data Descriptor .....	3-7
3-7	Input/Output Descriptor .....	3-8
3-8	Program Descriptor .....	3-8
3-9	Operand .....	3-9
3-10	Program Word .....	3-9
3-11	Program Register .....	3-10
3-12	Control Register .....	3-10
3-13	Literal Placed on Top of Stack .....	3-10
3-14	Obtaining Operand Directly From Program Reference Table .....	3-10
3-15	Constant Indexing .....	3-11
3-16	Variable Indexing .....	3-12
3-17	Multilevel Variable Indexing .....	3-13
3-18	Obtaining Descriptor Through Reference to Operand .....	3-13
3-19	Obtaining Descriptor by Referencing a Descriptor .....	3-14
3-20	Layout of Memory Module With Associated Registers .....	3-14
3-21	Variations in Format of Operand-Call and Descriptor-Call Syllables in Subroutine Mode .....	3-16
5-1	System Configuration Chart .....	5-2
5-2	The Console .....	5-2
5-3	Memory Module and Input/Output Channel .....	5-4
5-4	Input Information Flow .....	5-5
5-5	Output Information Flow .....	5-6
5-6	Storage Drum .....	5-7
5-7	Magnetic Tape Unit .....	5-8
5-8	Magnetic Tape Format .....	5-9
5-9	Line Printer .....	5-10
5-10	800 cpm Card Reader and Program Card Reader .....	5-11
5-11	Card Punch .....	5-12
5-12	Plotter .....	5-12
5-13	Message Printer/Keyboard .....	5-13



## TABLES

<b>Table</b>		<b>Page</b>
3-1	Function of Registers in B 5000 Processor.....	3-6
4-1	Input/Output Combinations With Maximum System.....	4-7
4-2	Input/Output Assignment.....	4-7
4-3	Memory Allocation.....	4-8
4-4	Base Locations of Program Reference Tables.....	4-8
4-5	Base Locations of Input/Output Descriptors.....	4-8
5-1	System Configuration Limits.....	5-1





**The B 5000 Information Processing System**

# SECTION 1

## INTRODUCTION

As the title implies, this manual contains a description in some detail of the B 5000 System and its operation. It is intended to provide technical information for those directly concerned with data processing—managers of data processing systems, programmers, and management personnel who are acquainted with the problems and concepts of electronic data processing. It is not intended as a teaching manual or programming primer. Without delving into such elements of computer design as circuitry and machine logic, this handbook explains how the B 5000 achieves its remarkable flexibility and efficiency through a comprehensive systems approach to problem solving.

The fact that the B 5000 works as a system—rather than only an advanced set of hardware—has determined the organization of this manual.

The source languages, ALGOL and COBOL, are presented first because they are the link between the user and the system, the statement of his problem and its ultimate solution. The next section, System Characteristics, discusses the way the B 5000 functions as a system and processes a program. Over-all coordination and control of processing, so important to total production through maximum use of the B 5000 components, is supplied by the Master Control Program, which is described next. Finally, the operational characteristics and features of the components which implement this unique system approach are specified in detail. Two appendixes provide comprehensive definitions of the programming languages, ALGOL and COBOL, including examples of their use. A glossary is included for reference to definitions of terms used with the B 5000 System.

The Descriptor is one of a series of technical publications which will give the user complete information on the equipment and its use. The B 5000 Concept Manual introduces the reader to the new design approach that integrates hardware and software—automatic programming and advanced operating techniques—into a unified package. This first edition of the Descriptor will be augmented by the publication of programming primers, reference manuals, and advanced programming manuals for ALGOL and COBOL.

The B 5000 was designed as a complete system, *combining* components and built-in programming aids, to bring the user simplified programming, ease of operation, and complete freedom of system expansion.

The programmer works in the language of his problem, either algebraic notation or English narrative statements. The B 5000 has compiler-oriented machine language and logic which accept the common languages of ALGOL and COBOL. Because the machine language of the B 5000 is similar to these problem languages, compilation time is reduced and object-program redundancies eliminated. The Processor operates on either single characters or complete words, alphanumeric or binary information, and the programs it uses are extremely compact. Since the B 5000 handles memory and input/output unit assignments, segmentation of programs, and subroutine linkages automatically, the programmer is freed of many arduous tasks and the likelihood of error is reduced. Correction of programs is done at the source-language level and is further simplified by integral debugging aids.

Operator intervention is nearly eliminated because complete management of the system is assumed by the Master Control Program, a comprehensive operating system that is recorded on the Storage Drum of the B 5000 System and provides simultaneous input/output operations and Multi-Processing. By controlling the sequence of processing, initiating all input/output operations, and providing automatic handling procedures to meet virtually all processing conditions, the Master Control Program can obtain maximum use of the system components at all times. Thus it achieves greater over-all production and efficiency. It assumes many functions that were formerly the responsibility of the programmer or operator, such as scheduling, allocation of memory, and assignment of input/output units. Because these functions are performed under central control, changes in schedule, system configuration, and program sizes can be readily accommodated.

It is this ability to control the processing pattern that provides the B 5000 with the potential for smooth growth. Since Processors, Input/Output

Channels, and core Memory Modules are independent of each other, the user may start with a basic system and only enough memory space to process his current programs. Large programs are processed in workable segments, so that the available memory is not filled with one job. As the volume of work swells, or larger programs are used, Memory Modules may be added. This increases Processor speed because more information can be processed concurrently. When input/output requirements exceed the capacity of the original equipment, a new Input/Output Channel can be added. Since the Master Control Program automatically adjusts

equipment assignments to use all available units, reprogramming is *never necessary*. Finally, a considerable growth in workload may warrant a second Processor, so that true parallel processing, as well as Multi-Processing, can be performed. The growth of the system in small increments—suited to the workload—is possible because the B 5000 is modular and because the programs are independent of the hardware. This Program Independent Modularity is the ability of the B 5000 to process programs on the equipment available, always making the most effective use of the system configuration, according to the needs of the application.

# SECTION 2

## PROGRAMMING LANGUAGE

### ADVANTAGES OF PROBLEM-ORIENTED LANGUAGES

This section explains the language used by the programmer to communicate with the B 5000 System. Three advantages of this language, as compared to conventional machine and compiler languages, are discussed:

1. Reduced programming time.
2. Reduced time to compile and run the object program.
3. Simplified debugging and program maintenance.

Programming for the B 5000 System is done in a problem-oriented language, rather than machine language. Thus the programmer is free to concentrate on the job of finding a logical solution and recording that solution in a language akin to his own.

Since most problems can be placed in one of two categories, computation or data manipulation, two distinct problem languages have been made available to the B 5000 programmer, one for each problem type.

### COMPUTATIONAL LANGUAGE (ALGOL)

The person responsible for solving a mathematical problem thinks about his subject in algebraic terminology. Many algebraic languages have been devised and have been in use for several years. However, experienced and capable computer users have become increasingly concerned in recent years about the number of these languages being produced. It has seemed that each new machine was accompanied by the announcement of a new automatic programming language.

An international conference of computer specialists was held to solve this problem. At this conference the groundwork was laid for development of a single language which would have three major characteristics:

1. It was to be independent of any computer so that programs written in this language would be available to all users regardless of the machine used.
2. It was to be as close to the thinking language of the programmer as possible to shorten the time required to formulate the solution on paper.

3. It was to be a language which incorporated every means conceivably needed for expressing solutions to problems of an algebraic nature.

A preliminary report on this conference was issued in December, 1958. The language was given the name ALGOL (ALGOrithmic Language). During 1959 the preliminary ALGOL specifications were studied by interested parties throughout the world. Informal meetings were held and correspondence was carried on by many computer-user groups in an effort to perfect this language.

Early in 1960, another international conference was held in Paris. As a result, ALGOL 60 was born. BURROUGHS has had a part in the development of ALGOL and is proud to include it as a part of the B 5000 Programming Language.

The following brief example may give the reader an idea of the use of ALGOL. A more complete explanation of ALGOL with a more complex example is given in Appendix A.

It is desired to add the quantities of A and B, and, if the sum (C) is greater than zero, stop the operation. When coding in ALGOL 60, this operation can be expressed in one simple statement, or program step, and that statement is like the thought process of the programmer.

**IF A + B > 0 THEN GO TO HALT;**

Also, in preparing this statement for input, it would be keypunched as it appears.

Compare this to the steps required when the same problem is coded in the machine language of another computer. It would be keypunched as follows:

```
0 0000 10 1000
0 0000 12 1001
0 0000 18 2000
0 0000 34 2050
```

### DATA-PROCESSING LANGUAGE (COBOL)

For defining a data-manipulation problem it would be ideal to use a language which is native to the pro-

grammer. Therefore, it has been a goal of the designers of automatic programming systems to provide as many English words as possible in the programming languages designed for business.

The same problems, however, that confronted scientific users became apparent in data-processing operations: too many languages were being developed and put into use. In May, 1959, a meeting was held in Washington with representatives from industry, government, and computer manufacturers in attendance. They agreed that the development of one common language tailored for business use was both desirable and feasible. There were three major requirements:

1. The need to translate existing data-processing problem solutions efficiently from one type of computer to another with minimum conversion costs.
2. The need for program documentation in a form allowing changes and additions with minimum time and expense.
3. The need for reducing the training period for programming personnel.

A report outlining initial specifications of a COmmon Business Oriented Language (COBOL) was published in April, 1960. These specifications represent COBOL 60.

On February 4, 1961, the COBOL Maintenance Committee—a group of 12 ADPM manufacturers, including Burroughs, and 10 interested industrial users—will complete the newest revision to the COBOL Specifications. The revised specifications will represent COBOL 61, which will be a programming language for the B 5000. The B 5000 COBOL translator is designed so that future modifications such as COBOL 62 can be readily incorporated.

With COBOL, as with any language, certain rules of construction must be followed. The features of COBOL are outlined in Appendix B, together with an example. However, to give an indication of the nature of COBOL, a simple example is given here.

It is desired to update an employee's annual FICA immediately if his monthly FICA is greater than or equal to the average. If it is less than the average, it is to be given special consideration before updating the annual balance.

Using COBOL, this situation can be handled by one procedure, and would be handled as follows:

```
IF MONTHLY-FICA LESS THAN 16.00 GO  
TO SPECIAL-FICA; OTHERWISE ADD  
MONTHLY-FICA TO ANNUAL FICA.
```

Using the machine language of another computer, the same situation would be coded as follows:

```
0 0000 10 1000  
0 0000 18 2000  
0 0001 34 2050  
0 0000 19 1010
```

The BURROUGHS B 5000 Programming Language, therefore, includes the two most widely accepted computer languages yet devised, ALGOL and COBOL. They are usable in the form designated by their originators and are not just another manufacturer's modification. The use of ALGOL and COBOL will greatly reduce the time required to program the problem and will result in more B 5000 production per hour. Extensive studies, comparing the time required to code a variety of problems in machine language with that to program them using ALGOL, have shown a reduction in the over-all time in a ratio of approximately 20 to 1.

### **REDUCED COMPILING AND RUNNING TIME**

Another important factor contributing to the Through-Put of the B 5000 is the greatly increased compilation speed of its translators. Programs that once required machine compilation time from 30 minutes to nearly two hours can be handled by the B 5000 in 30 seconds to a minute or two.

Not only does the B 5000 translate fast, but it produces a very efficient object program. In the past, an experienced programmer could see at a glance many superfluous instructions in a compiled program. This is not true of a B 5000 object program.

### **SIMPLIFIED DEBUGGING AND PROGRAM MAINTENANCE**

A serious obstacle to the completion of a program is often the amount of time required for debugging. The B 5000 programmer need not concern himself with the easily made and hard-to-find errors created in the manual process of translating a logical solution into detailed machine instructions. He looks only at the logic of the problem, which is defined in an understandable language.

The difference in complexity of debugging requirements can best be illustrated by an example. Suppose that a reporter who does not understand Japanese is required to write a story for Japanese distribution. Upon completion of the task a Japanese proofreader informs him only that the story is not correct. The reporter would then be required to check each character of his story with the aid of an English-Japanese dictionary, a task which could prove as time consuming as the writing itself.

On the other hand, suppose that an interpreter were available to inform the reporter of factual errors in the story due to the language difference. The reporter then need only review his English copy to correct the specified facts.

The B 5000 incorporates a translator analogous to the interpreter. The programmer need never know computer language to write an effective program for it.

Program revisions can be as tedious as debugging. Such revisions may be dictated by an altered problem definition, often occurring many weeks after completion of debugging. To facilitate making these changes it is desirable to document the program with logical flow charts, file layouts, and solutions to the problem. With the B 5000, the documentation is readily understandable—not only to the programmer but to anyone concerned with the problem. Special-purpose flow charts slanted toward particular machine characteristics or the machine language instructions themselves are not required. Occasional changes to the program are accomplished in problem-oriented language. The changes are concerned with the logical solution, not the machine language instructions.

## **PROGRAMMING ADVANTAGES OF THE B 5000 SYSTEM**

### **REDUCED PROGRAMMING TIME**

The BURROUGHS B 5000 is designed to make use of the best programming language presently available, a problem-oriented language which includes the most recent developments in this field, ALGOL 60 and COBOL. Because ALGOL and COBOL were devised by top people in their respective fields, it is reasonable to expect that these languages will not be replaced for years to come. ALGOL and COBOL have already received wide acceptance and their use is fast becoming universal. It is therefore possible to create a program library which will have a life span beyond that of the particular computer equipment being used at the time. The exchange of programs between different users is also practical.

The design characteristics of the B 5000 contribute significantly to greatly reduce total programming time. The programmer need not be concerned with such things as actual memory locations, specific input and output unit and magnetic tape unit designation. In the B 5000, these details are handled automatically while the program is being run. The same object program can be run without recompilation even when the system configuration changes. These housekeeping details are automatically handled by the Master Control Program.

## **REDUCED COMPILING AND RUNNING TIME**

The BURROUGHS B 5000 is capable of very fast compilation speeds and efficient object programs because the system has been designed to take advantage of the latest automatic programming techniques.

Recent BURROUGHS compilers have made use of these techniques, resulting in compilation speeds as much as 50 times faster than those of other compilers.

**Conventional Compiler Functions.** The operation of compilers must be considered in order to understand the advantages of these BURROUGHS compiler techniques. A compiler is a program able to translate from one language to another, i.e., from a language which is intelligible to a programmer to a language which is understood by an electronic computer. Programming languages have changed and have begun to approach the thinking language of the programmer. These improvements, however, have resulted in the programming language looking less and less like the machine language. This means that the compiler requires more and more time to perform the translation functions required to produce an object program in machine language.

The conventional compiler does three basic things:

1. Examines the programmer's language.
2. Creates an intermediate language.
3. Produces machine language.

First, it examines the programmer's language (source program), one character at a time and establishes a dictionary of separate entities, such as identifiers, numbers, and variables. The compiler is able to do this by recognizing separator and operator symbols of the source language such as colons, equal signs, and parentheses. For example, if the programmer had written somewhere in his program:

START:  $A \leftarrow 7 \times (B + C)$

the compiler would begin to scan with the letter S. It would then encounter the letters T, A, R, and T in that order. When it recognizes the colon as the next character, the compiler knows that START is a separate item and would add it to the list. Next the compiler encounters A followed by an arrow, so it would add A to the list. Similarly for 7 and B and C. The dictionary contains, in addition to the item's name, its type and an indication of its location.

Secondly, the compiler produces an intermediate language which essentially breaks up the program into separate operational groupings. For instance, in the above example, the following equation must be evaluated:



$$A = 7 (B + C)$$

Earlier conventional compilers would break this equation up into two operations:

$$\begin{aligned} A &= 7 \times T \\ T &= B + C \end{aligned}$$

(where T stands for some temporary storage location)

A temporary location is established each time an enclosure symbol is encountered during the scanning process. When a left parenthesis is found in an arithmetic expression, for example, it is immediately known that a single value is to be defined. The set of operations for computing this value will be terminated by the associated right parenthesis. When the termination will occur is still unknown. Enclosure symbols may be nested and expressions quite complex. Therefore, this problem has been handled by creating many temporary locations. Later an attempt must be made to eliminate as many as possible and the intermediate language condensed.

Finally, the actual machine language is produced through the use of generators within the compiler itself. Because these generators (subroutines) are of a general nature they must be constructed to take care of all possible cases. Therefore, the resulting machine language (object program) contains unnecessary instructions. For instance, the above example might result in the following:

LOAD	B	
ADD	C	Add operation generator.
STORE	T	
LOAD	7	
MULTIPLY	T	Multiply operation generator.
STORE	T	
LOAD	T	Store operation generator.
STORE	A	

An experienced programmer would use fewer instructions to arrive at the same solution:

LOAD	B
ADD	C
MULTIPLY	7
STORE	A

Therefore, the object program produced by earlier compilers would take, in most cases, considerably longer to run than one written by an experienced programmer.

**Burroughs Compiler Techniques.** The first task of a compiler, that of scanning the source program, is also a part of the B 5000 translators. It is interesting to note, however, that in a conventional

compiler this process is just a first step. On the B 5000, one pass through the programmer's language is all that is required.

The effectiveness of the BURROUGHS compiler technique is most readily recognized in the second phase of compilation. The example  $A = 7 (B + C)$  as described illustrates that parentheses caused conventional compilers to do a great deal of extra work in setting up temporary storage locations. Nonetheless, parentheses are needed in the source language to eliminate ambiguities. For instance, the example could be interpreted another way if there were no parentheses. Namely:

$$A = 7B + C$$

This problem of ambiguity was also quite perplexing to philosophers who were considering logical propositions until a Polish logician, J. Lukasiewicz, developed a notational system which did not require enclosure symbols. His scheme is commonly called Polish notation. The BURROUGHS technique produces an intermediate language using Polish notation concepts.

**POLISH NOTATION.** The essential difference between Polish notation and conventional notation is that operators are written to the right of a pair of operands instead of between them. For example, the conventional  $B + C$  would be written  $BC+$  in Polish notation. Looking again at the example,  $A = 7 (B + C)$ , it could be written as follows:

$$BC + 7 \times A =$$

Any expression written in Polish notation is called a Polish string. In order to fully understand this concept, the rule for evaluating a Polish string should be known.

The rule can be summarized in a few steps:

1. Scan the string from left to right.
2. Remember the operands and the order in which they occur.
3. When an operator is encountered do the following:
  - a. Take the two operands which are last in order;
  - b. Operate upon them according to the type of operator encountered;
  - c. Eliminate these two operands from further consideration;
  - d. Remember the result of (b) and consider it as the last operand in order.

Following this rule through the Polish string,  $BC + 7 \times A =$ , step by step would result in:

STEP	THE SYMBOL BEING EXAMINED	SYMBOL TYPE	OPERANDS BEING REMEMBERED AND THEIR ORDER OF OCCURRENCE	OPERATION TAKING PLACE	RESULT OF OPERATION
a	B	Operand			
b	C	Operand	1 B		
c	+	Add Operator	2 C 1 B	B + C	(B + C)
d	7	Operand	1 (B + C)		
e	×	Multiply Operator	2 7 1 (B + C)	7 × (B + C)	7 (B + C)
f	A	Operand	1 7 (B + C)		
g	=	Replace Operator	2 A 1 7 (B + C)	A = 7 (B + C)	

The result is that A has assumed the value  $7(B + C)$ . If the reader is interested in testing his grasp of Polish notation, a conventional algebraic expression is given below. An equivalent Polish string may be found at the bottom of the page.

$$y \leftarrow (w + i + t) (p - q) / z$$

**OPERAND STACK.** Polish notation is used as the formulation principle of the B 5000's intermediate language. Very effective techniques have been devised for use in its production. One of these involves the utilization of two reserved areas. One holds operands and is called an Operand Stack; the other contains operators.

As the B 5000 translator scans the source program, it places any operands encountered in the Stack, operators, and enclosure symbols in their reserved area. To show how these areas function, the example given as an exercise above will be used. Figure 2-1 shows a simplified flow chart of the translation process, presented only for illustrative purposes. In Figure 2-2, the reserved memory contents are shown at each stage of the scanning process. It will be noticed that only the essentials are retained in memory. The need for temporary storage locations is eliminated.

In summary, the BURROUGHS B 5000 compiler is considerably faster than conventional compilers dur-

ing the second phase of the compilation process because of three things:

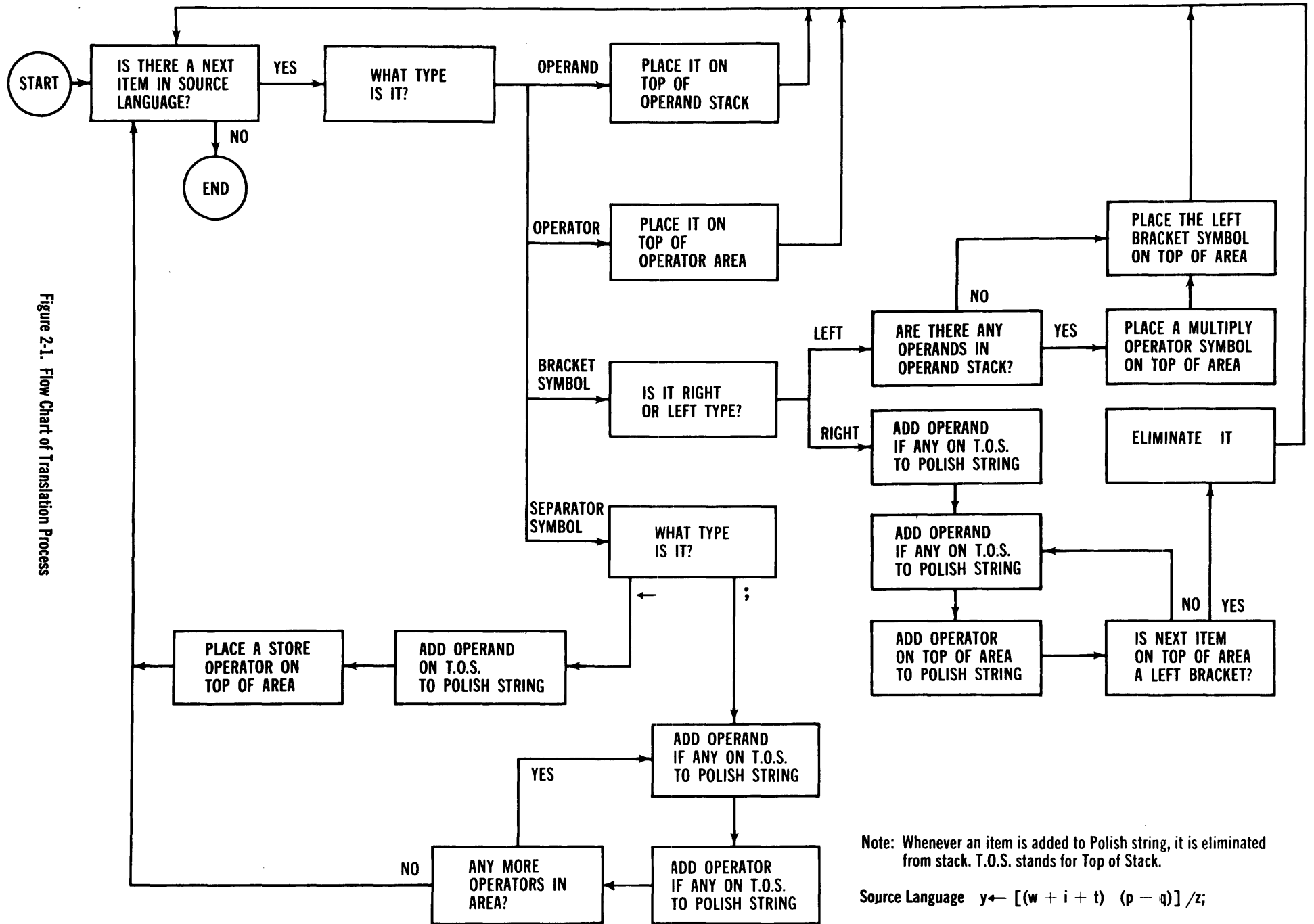
1. It produces an intermediate language which has formulation characteristics very similar to that of the source language, requiring only a simple translation.
2. This intermediate language is free of ambiguity, yet is composed of a minimum number of symbols.
3. The methods used to form this language (Polish notation), such as the stack concept, have proved to be extremely efficient.

The third phase of the normal compilation process is practically eliminated, not by a compiler technique but through the revolutionary design of the BURROUGHS B 5000. The structure of the B 5000 machine language is patterned after that of Polish notation. Therefore, for all practical purposes, the B 5000 translator has produced the desired machine language after the second phase.

Another factor contributing to Through-Put is the running speed of the *object* program. When a B 5000 program is running, it is in essence evaluating a Polish string. Referring to the example of such an evaluation process, for the string  $BC + 7 \times A =$ , it can be seen that an operand stack would be ideal for

Polish string equivalent:  $ywi + t + pq - xz / =$

Figure 2.1. Flow Chart of Translation Process



Note: Whenever an item is added to Polish string, it is eliminated from stack. T.O.S. stands for Top of Stack.

Source Language  $y \leftarrow [(w + i + t) (p - q)] / z;$

SOURCE LANGUAGE ITEM BEING EXAMINED	CONTENTS OF STACK FOR OPERANDS	CONTENTS OF AREA FOR OPERATORS, ETC.	INTERMEDIATE LANGUAGE (POLISH STRING) BEING BUILT	SOURCE LANGUAGE ITEM BEING EXAMINED	CONTENTS OF STACK FOR OPERANDS	CONTENTS OF AREA FOR OPERATORS, ETC.	INTERMEDIATE LANGUAGE (POLISH STRING) BEING BUILT
y	y			p	p	( x [ =	yti+w+
←		=	y	-	p	- ( x [ =	yti+w+
[		[ =	y	q	q p	- ( x [ =	yti+w+
(		( [ =	y	)		x [ =	yti+w+qp-
w	w	( [ =	y	]		=	yti+w+qp-x
+	w	+ ( [ =	y	/		/ =	yti+w+qp-x
i	i w	+ ( [ =	y	z	z	/ =	yti+w+qp-x
+	i w	+ + ( [ =	y	:			yti+w+qp-x/=
t	t i w	+ + ( [ =	y				
)		[ =	yti+w+				
(		( x [ =	yti+w+				

NOTE: The reader will notice that the operands p and q are transposed in the final Polish string produced. The subtract operator in the B 5000 is defined so that these operands will be ordered properly during running time. The Polish string built by the B 5000 also is one which minimizes the contents of the operand stack during object time.

Figure 2-2. Stages of the Scanning Process

retention of the "operands being remembered." Their order in the stack is automatic when using the "top-of-stack" concept. This concept is an ordering process which is based on the last-in-first-out principle. The information placed last in the stack is the next to be removed and used. Therefore, the operands needed for an arithmetic operation are automatically located on the "top of the stack." These features plus many others associated with the evaluation of Polish strings are incorporated into the design characteristics of the B 5000. Further insight into the contribution of the B 5000's logic to Through-Put is given in a later section exclusively devoted to that subject.

### SIMPLIFIED DEBUGGING AND PROGRAM MAINTENANCE

The job of debugging a B 5000 program is made easier for three distinct reasons: there are fewer mistakes to find; most errors are automatically dis-

covered; and debugging is carried on in programming language. set of instructions includes a great many possible combinations of these digits, the possibility of finding a transcription error is remote. Therefore, a program written for this purpose is not worth the effort. In the B 5000 programming system, however, most syntactic errors are obvious and will be discovered automatically during the translation process. For instance, in the problem  $y \leftarrow [(w + i + t) (p - q)]/z$ ; assume the keypunch operator had transposed two symbols,  $i$  and  $+$ , so that the input document read as follows:  $y \leftarrow [(w + \underline{+} i t) (p - q)] /z$ . The B 5000 translator would recognize this mistake immediately because it knows that no two arithmetic operator symbols can be adjacent to one another. It would so inform the programmer.

If, on the other hand, machine language was being used and the keypunch operator made a transposition error, in the address of  $i$  for instance, it would be undetectable. For example:

MNEMONIC FORM	MACHINE LANGUAGE DESIRED		MACHINE LANGUAGE ACTUALLY KEYPUNCHED	
	OP CODE	ADDRESS	OP CODE	ADDRESS
Load w	15	7593	15	7593
Add i	13	8216	13	<u>2816</u>
Add t	13	1699	13	1699

covered; and debugging is carried on in programming language.

The use of a problem-oriented language introduces fewer errors into the document to be read by the computer than the use of machine language. Every problem solution must first be formulated on paper in the language of the programmer. Translating this solution to machine language is monotonous and a major source of errors. Although this job is ideally suited to computers, their use has been too costly. The BURROUGHS B 5000 makes this feasible, assuring fewer errors in input documents.

A conventional machine language instruction consists of a fixed number of digits. Since the complete

No reasonable program could be devised for detecting this error.

Finally, when debugging is found to be necessary, the B 5000 provides tools and clues which are easily understood by the programmer. He may write special debugging statements in the B 5000 programming language and place them temporarily at the beginning of his input deck. As a result of these debugging statements, the programmer is provided with information about those characteristics in which he has indicated an interest. This information is given in his programming language so that an immediate comparison with his program is possible.

# SECTION 3

## SYSTEM CHARACTERISTICS

### GENERAL DESCRIPTION

When a program has been written in one of the languages acceptable to the BURROUGHS B 5000 System—either ALGOL 60 or COBOL 61—it is punched on cards and loaded into the system. The object program is then compiled and recorded on a magnetic tape called the program tape or punched on program cards. The compiled program consists of program segments and an associated Program Reference Table. The length of the program segments varies and is determined by the compiler according to the over-all length and nature of the problem. Program parameters are also produced during compilation. These specify the input-output devices required, the amount of storage required for the program itself and for its associated data, and other information necessary for processing the program.

Figure 3-1 shows the transformation of the source program into an object program.

### STORAGE DRUM

The BURROUGHS B 5000 System includes a magnetic drum as well as core storage. Part of the drum is reserved for storage of the compiler and the Master Control Program. This area is not available to the user. The remainder of the drum is used for program segments and their associated Program Reference Tables, as illustrated in Figure 3-2.

### CORE MEMORY

The Master Control Program is loaded into memory initially by depressing the Program Load button.

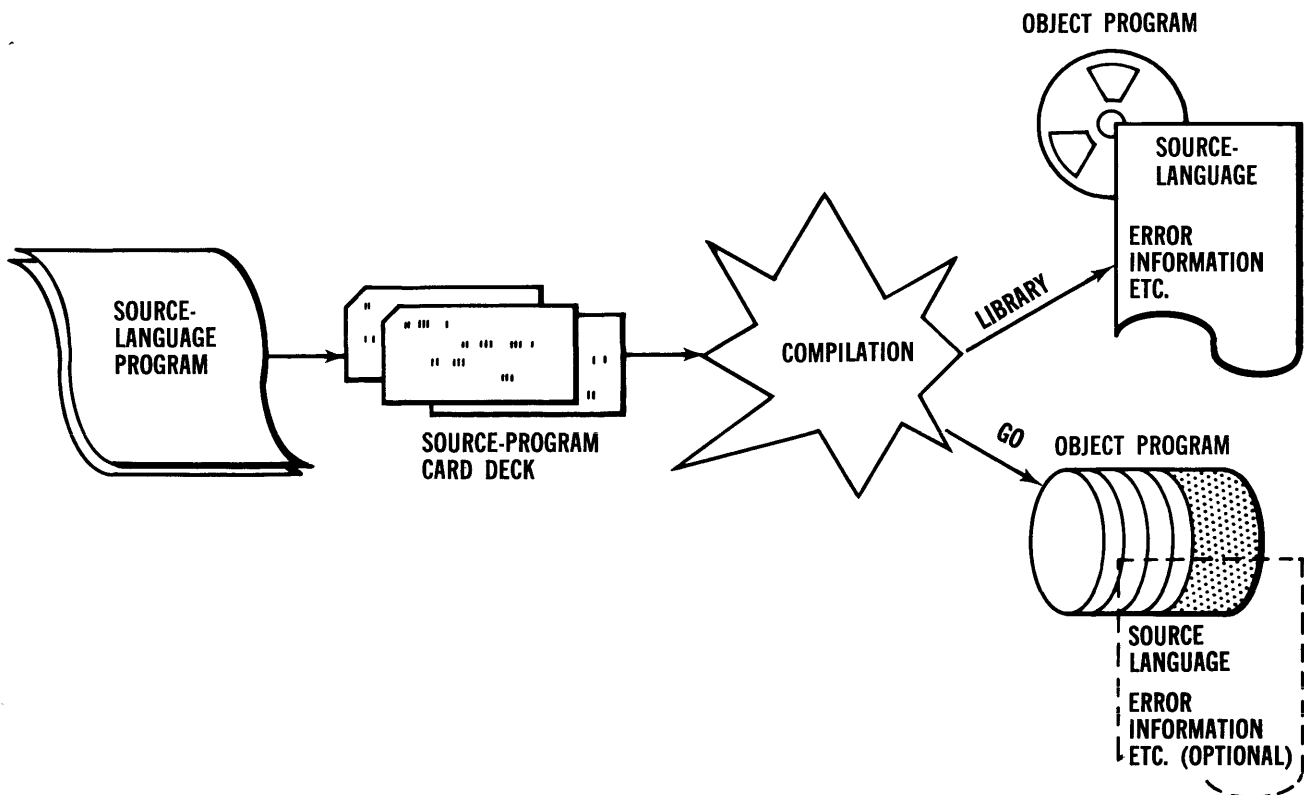


Figure 3-1. Compilation of Source Program

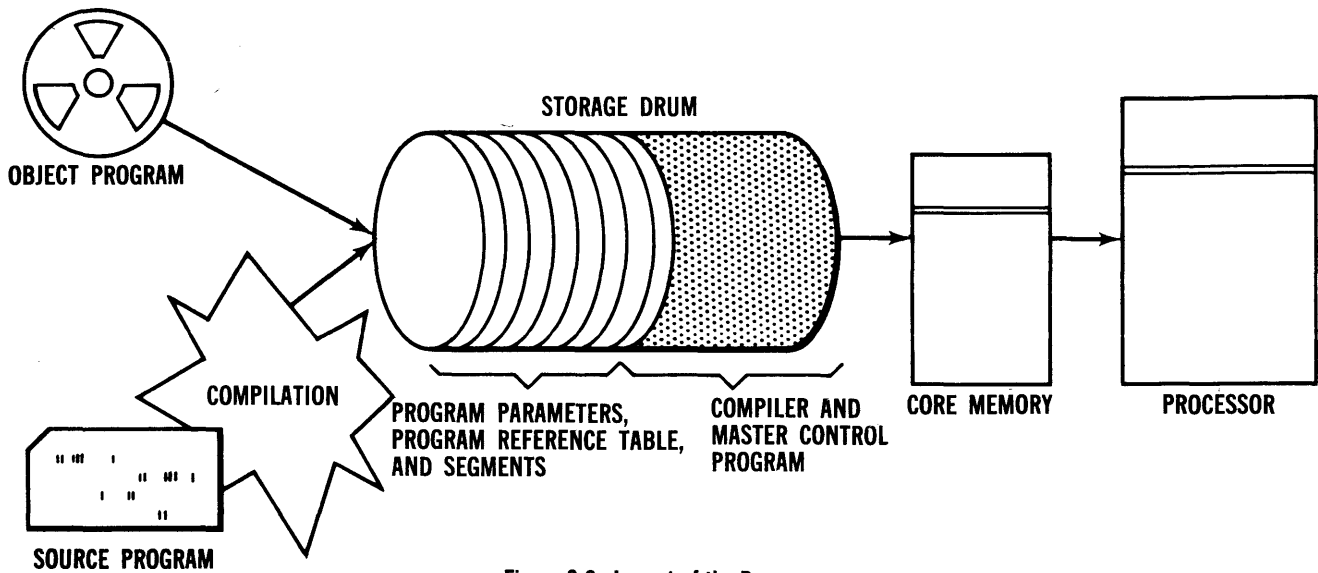


Figure 3-2. Layout of the Drum

Its function is to select the scheduled programs from either the program tape or punched cards and load them onto the drum. Then, on the basis of the program parameters supplied, the Master Control Program allocates memory space for each program to be processed. Next, it loads the required program segments and associated Program Reference Table for each job. At this point the Program Reference Table and program segments pertaining to each job

are contained in memory. Other areas of memory have been allocated to contain the Stack and input/output information, as shown in Figure 3-3.

**WORDS**

Information is stored in the Program Reference Tables and program segments in words containing 48 bits of information. There are three types of words, each having a particular form and function:

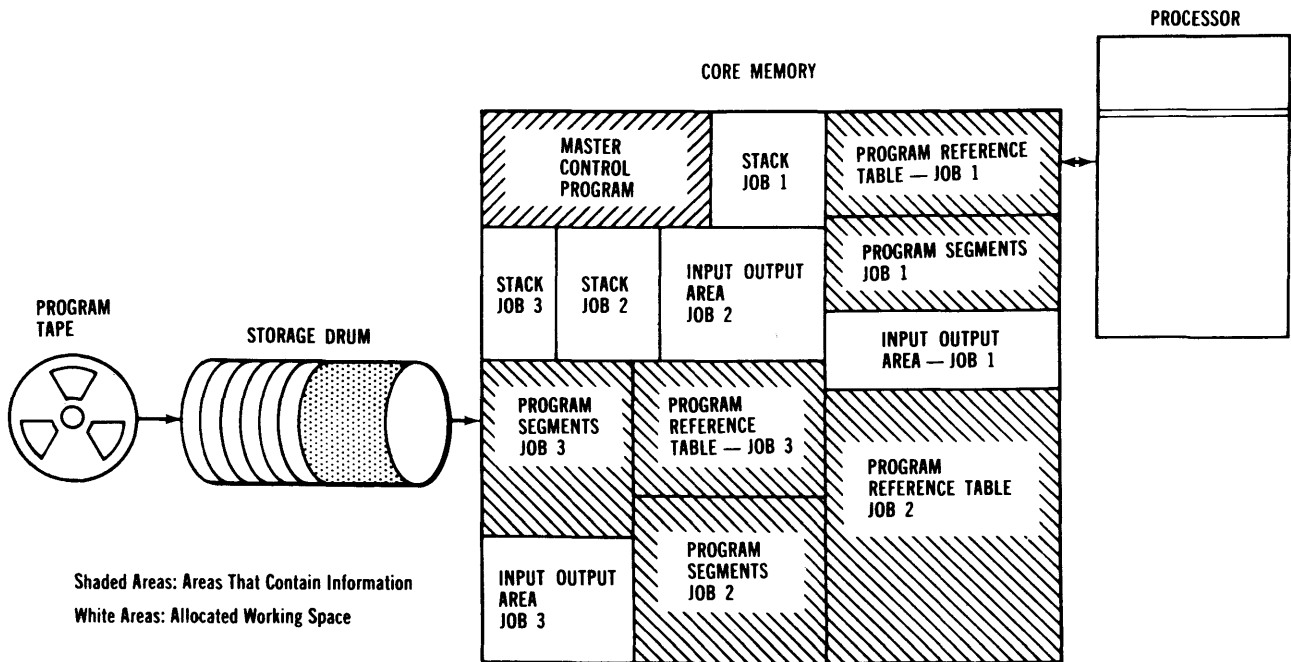


Figure 3-3. Allocation of Core Memory

descriptors, operands, and program words. The descriptors and operands make up the Program Reference Table, and program words are found in the program segments. Each type is introduced below and discussed in further detail.

### PROGRAM REFERENCE TABLE

The Program Reference Table is an area of storage containing control words that are used to *locate* data. They may also *describe* the type of input/output operation to be executed. These control words, known as descriptors, provide the base or starting address of a program segment, file of records, input/output area, or subroutine. When they specify input/output operations, they designate the unit to be used, format, and the location and amount of information to be transferred.

Operands are also located in the Program Reference Table and provide direct access to single values. This type of word contains the *actual* value instead of the location of that value.

### PROGRAM SEGMENTS

A program segment, composed of program words, is a self-contained group of steps that does a portion of the processing for a job. Each program word is divided into four syllables. And there are four *types* of syllables used in program words: operand-call, descriptor-call, literal, and operator. Essentially these are the instructions for processing the data referenced by the Program Reference Table.

### THE STACK

This is an area of memory set aside for temporary storage of information. It is associated with the A and B registers, the arithmetic registers of the B 5000 System. The registers form the first and second locations of the Stack, and information transferred to or from the Stack passes through them. Its operation is much like that of the stack used to compile the Polish notation described in Section 2.

The last piece of information placed in the Stack is the first to be accessed again. Previous entries in the Stack are "pushed down" when new information is added; that is, the contents of the A and B registers are shifted into storage locations allocated for Stack use. As more space is required, a new storage location (addresses are used in ascending order) is added to the Stack, receiving the information shifted out of the registers. Figure 3-4 illustrates the shifting of information as new data is brought to the Stack for processing.

The term "top of the Stack" refers either to the first Stack location in memory or to the A or B register, depending on the placement of the data. If an operand is transferred from the Program Reference Table it is placed in the A register, which is, at that moment, the top of the Stack. The results of many operations, however, are placed in the B register, leaving the A register empty; in this case, the B register becomes the top of the Stack. Under some conditions both the A and B registers are cleared, and then the first memory location in use is the top of the Stack. Whenever a sequence of operations requires information stored in the Stack memory locations, it is automatically shifted into the registers where it can be processed.

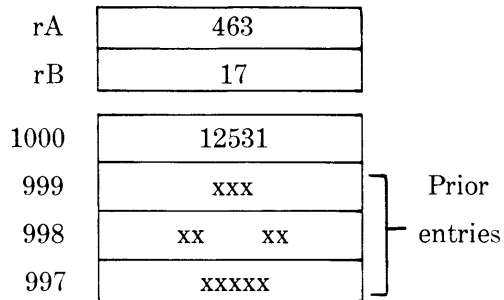
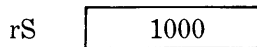
The location currently in use as the top of the Stack in memory is recorded in the S register. If new locations are added to hold information brought to the Stack, the S register is counted up. When an operation causes items in the Stack to be removed, those locations are removed from the Stack until needed again, and the S register setting is counted down by the corresponding amount. The total number of locations used varies according to the processing sequence. Referring to Figure 3-4, assume that a series of operations uses the information stored in the A and B registers and in Stack locations 1002 and 1001 (stage 3 in the example). The result of the operations is held in the B register, and the S register is set to location 1000. Figure 3-5 shows the condition of the Stack at this point. (Note that the A register is empty, and therefore the B register has become the top of the Stack; if both registers were empty, location 1000 would be the top of the Stack.)

The allocation of memory in Figure 3-3 shows three areas set aside as Stacks for Multi-Processing of three different jobs. Since the Stack is merely a flexible working area, its actual location is unimportant. The Stack in use at any given moment is associated with the A and B registers, and the memory address at the top of the Stack being used is recorded in the S register. When the Processor switches from one job to another, any information held in the A and B registers is automatically shifted into memory locations prior to transfer of control. The current S register setting is always preserved for re-entry to the job.

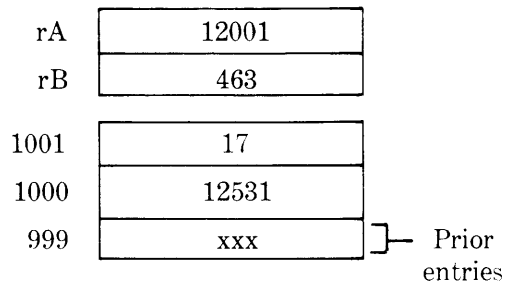
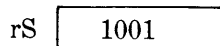
The Master Control Program, described in Section 4, controls the processing sequence of various jobs. It allocates Stack memory space according to the number and size of programs in memory at one time. Switching from job to job, and therefore from Stack to Stack, is also supervised by the Master Control Program.



1. The A register contains the number 463 and the B register 17. The top of the Stack at this stage is location 1000, and its contents are 12531.



2. Execution of a program syllable causes new number to be placed in the A register, causing the number 463 to shift into the B register, and 17 to be placed in location 1001, which becomes the first location of the Stack in memory.



3. Another quantity is brought into the Stack and at this stage the original contents of the A and B registers have been completely replaced by new information.

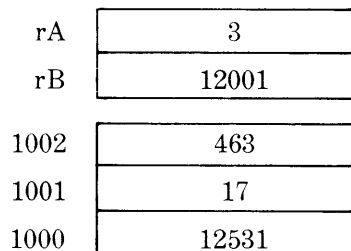
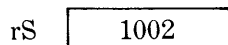


Figure 3-4. Shifting of Information in the Stack

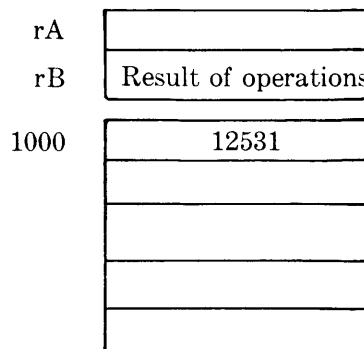
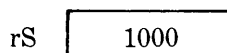


Figure 3-5. Condition of Stack and Associated Registers After Information Has Been Removed

To summarize, the Stack is the means by which the Processor has access to the information being processed. Operand-call and descriptor-call syllables in the program segment cause information to be transferred from either the Program Reference Table or referenced data arrays and placed in the Stack. Operator syllables from the program then cause this information to be processed in the specified manner. All data manipulation and execution of computer operations affects information that is brought to the Stack.

## INPUT/OUTPUT AREAS

These are temporary storage areas. Information is read into an input area, and thence transferred to the Stack for processing. After the specified operations have been performed, the results are stored in an output area until an Input/Output Channel is free and they are recorded on magnetic tape, punched cards, or printed forms.

## MEMORY ADDRESSES

It should be noted that none of the descriptors created during compilation contains a base or starting address. Thus, every program is completely relocatable. The Master Control Program assigns base addresses each time the program is run for all data arrays referenced by the Program Reference Tables; input/output areas; program segments; and the Stack. Base addresses are recorded in the Program Reference Table, which is used throughout processing. Thus changes in control are accomplished automatically and without programming effort.

## PROCESSING

The programs in memory are now ready for execution by the Processor. This component houses the arithmetic and logic control elements of the system. The details of processing are described in this manual only for the information of the reader. *The programmer and user are never required to know about the internal functions of the Processor since machine-language coding is not used with the B 5000 System.*

## MODES OF OPERATION

The Processor operates in four modes: Arithmetic or normal mode, Subroutine mode, Data Manipulation or editing mode, and Control mode. The first three are defined and described fully in this section. The Control mode, which is in effect whenever the Master Control Program is being used, is discussed in Section 4, Master Control Program.

A number of registers are contained in the Processor. Their function varies according to the mode of opera-

tion in effect. Table 3-1 gives a brief summary of register operation in each mode.

The function of the Arithmetic and Subroutine modes is similar in that they are used to carry out computational processes as specified by the source-level program statements. The characteristics of the Program Reference Table, Stack operation, and the registers are therefore much alike in these two modes. The rearrangement or editing of input data, internally generated information, and output data is accomplished in the Data Manipulation mode. It differs from arithmetic and subroutine operation in use of the registers and in the pattern of information transfer between memory and the Processor.

In the material to follow, the characteristics of the Program Reference Table, descriptors, operands, and program syllables are described in detail as they function in the Arithmetic mode. The section on the Subroutine mode describes those features which distinguish it from the Arithmetic mode and presents typical subroutine entry and exit sequences. Examples are used throughout to demonstrate the execution of a program syllable and its effects upon pertinent functional components—the Program Reference Table, Stack, and registers.

Finally, the operating characteristics and language of the Data Manipulation mode are introduced. It should be kept in mind that the programmer never has to create or use the editing strings; these are normally created as a result of descriptions of the input and output supplied by the programmer.

## OPERATION DURING ARITHMETIC MODE

### INTRODUCTION

The Arithmetic mode is concerned with the execution of program words. As a program segment is processed, each word is brought to the P register. The four syllables of the words are executed sequentially, causing operations to be performed upon information located by means of the Program Reference Table. The descriptors stored in this table are used to locate the data to be processed, temporary storage locations, subroutines needed by the program, or other segments as they are needed. Operands which provide single values for direct use may also be stored there.

Execution of a syllable (if operand-call, descriptor-call, or literal) causes a word to be placed in the Stack or (if operator) operates on words already in the Stack.

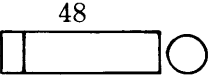
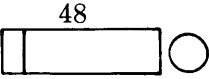
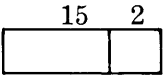
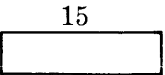
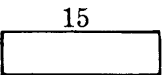
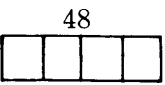
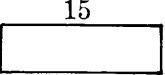
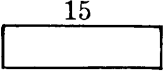
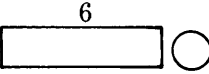
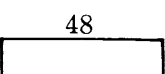
It is in the Stack, which is associated with the arithmetic (A and B) registers of the system, that operations are actually performed. Output is produced according to specifications provided by output descriptors also found in the Program Reference Table.

### PROGRAM REFERENCE TABLE

This is a 1024-word area of core memory which pro-

vides a dictionary of all information required by the program being processed. One Program Reference Table is allocated for each program being processed concurrently. The table is constructed of descriptors and operands during compilation. When the Master Control Program schedules a certain program, the program tape or card deck containing the Program Reference Table and program segments is loaded

Table 3-1. Function of Registers in B 5000 Processor

NAME	ARITHMETIC MODE	SUBROUTINE MODE	DATA MANIPULATION MODE
A: 	Arithmetic register/top of Stack.	Same as arithmetic mode.	Source register, holds one word of data to be edited.
B: 	Arithmetic register/second word in Stack.	Same as arithmetic mode.	Destination register contains edited data that is being formed into a word.
C: 	Contains address of next syllable to be processed.	Same as arithmetic mode.	Same as arithmetic mode.
F: 	Not used.	Contains the location in Stack where return point from S-R is stored.	Source-address register, used to obtain the current word to be edited.
M: 	Indicates the location in memory (actual memory address) to which reference is being made.	Same as arithmetic mode.	Destination-address-register contains the address where the contents of rB will be stored.
P: 	Holds four-syllable word currently being processed.	Same as arithmetic mode.	Same as arithmetic mode.
R: 	Contains the base address of PRT currently being referenced.	Same as arithmetic mode.	Same as arithmetic mode.
S: 	Contains the last location of the Stack used (except rA and rB).	Same as arithmetic mode.	Same as arithmetic mode.
T: 	Not used.	Not used.	Count register used to count number of characters defined by manipulation syllable.
X: 	Extension of rA, used only for multiplication and division, as well as some special functions.	Same as arithmetic mode.	Same as arithmetic mode for the special functions.

onto the magnetic drum. The Master Control Program allocates areas in core memory for the table, segments, input/output storage, and the Stack and transfers the relevant information from drum to memory.

**Descriptors.** Each descriptor requires one word, consisting of keys used to locate data or to specify input/output operations. This indexing procedure permits all programs for the B 5000 System to be completely independent of actual storage addresses and the hardware configuration, provided, of course, that a minimum system is operative. The facility to relocate a program easily and automatically means that it can be segmented by the compiler into efficient working lengths and that interruption of processing does not result in loss of time or effort. Processing resumes automatically from the point of interruption, even if the program has been removed from memory and loaded into a new area in the meantime.

At the time of compilation, descriptors contain identification labels and information about the size of the data record identified. Every time the Program Reference Table is loaded into core memory, the Master Control Program supplies core base addresses for each group of data referenced in the table. The term "descriptor" implies the job performed: to describe the size and location of a group of information required by a program. Because the actual location of this information is determined and recorded each time the program is loaded, all programs and associated data, working areas, and input/output operations can be relocated automatically according to current processing conditions. The programmer is never concerned with storage or input/output unit assignments.

Three types of descriptors are used: data descriptors, input/output descriptors, and program descriptors. The following paragraphs discuss their respective formats, operating characteristics, and use.

**DATA DESCRIPTORS.** Any sort of data array—for example a table, records from a master file, working areas for intermediate results, vectors, or sets of coefficients—may be indexed by a data descriptor. The format of this descriptor is shown in Figure 3-6.

The identification and size fields are supplied during compilation; the drum number and drum address are inserted by the Master Control Program if and when the information is loaded onto the drum. The core address field is filled in by the Master Control Program when it has reserved space for the data according to the specification given in the size field.

The presence bit indicates whether the data is currently available in core memory or still on the drum. It is automatically set to 0 when information is transferred to core memory. When an object program refers to an element of an array, the descriptor of that array is read from the Program Reference Table. If the presence bit is set to 0, the array is located in storage at the core address contained in the descriptor. If the presence bit is 1, an interrupt condition automatically occurs; the Master Control Program transfers the array from drum to core, setting the presence bit of the descriptor to 0, filling in the core address of the descriptor, and returns control to the object program.

**DRUM DESCRIPTOR.** This word is a variation of the data descriptor and has the same format (refer to Figure 3-6). It is used when the Master Control Program assumes control in order to transfer data from drum to core, or from core to drum. It differs from the data descriptor in the setting of the presence bit and the core address fields. The drum descriptor is defined here because it is discussed in connection with the operation of the Master Control Program. It is used during normal core-drum transfer and whenever the MCP requires a core-to-drum or drum-to-core transfer.

FLAG	IDENTIFICATION	PRESENCE BIT	DRUM NUMBER					SIZE FIELD (NUMBER OF WORDS REFERENCED)	DRUM ADDRESS	CORE (BASE OR STARTING) ADDRESS
------	----------------	-----------------	----------------	--	--	--	--	--	--------------	---------------------------------------

Figure 3-6. Data Descriptor

FLAG	IDENTIFICATION	UNIT	SIZE FIELD (NUMBER OF WORDS REFERENCED)	TYPE OF OPERATION	TAPE CHARACTER FORMAT — PRINTER CARRIAGE CONTROL INFORMATION	CORE (STARTING OR BASE) ADDRESS
------	----------------	------	---	----------------------	---	--

Figure 3-7. Input/Output Descriptor

**INPUT/OUTPUT DESCRIPTORS.** These have the format shown in Figure 3-7. The compiler provides information for the identification and size fields. The size field specifies the number of words to be transferred to or from core memory. Tape format and printer carriage control specifications are filled in by the compiler. Once the Program Reference Table is loaded, and before control can be transferred to the object program, the other fields of the word must be completed.

The Master Control Program scans the list of input/output descriptors, assigning unit numbers according to the hardware available and the units on which the operator has mounted the tape reels. (This process is discussed in detail in Section 4, Master Control Program.) Corresponding input/output areas in core memory are reserved for each descriptor on the basis of the information provided by the size field, and the core address is inserted in the descriptor.

The field titled "information for machine use" designates the type of operation to be performed. For example, any of several magnetic tape operations may be specified: read, write, backspace, rewind, or erase. Reading and punching of cards, line printing, or plotting may be indicated. The compiler produces appropriate specifications according to the source-language instructions.

**PROGRAM DESCRIPTORS.** These are used for transfer of control to program segments or subroutines. Program descriptors are located in the Program Reference Table. There is a descriptor for each segment of the object program and one for every subroutine called for. The format of the program descriptor is shown in Figure 3-8.

The identification and size fields are supplied during compilation, and the Master Control Program inserts the drum designation and base address when it loads the segments or subroutines onto the drum.

Program segments are called in only as required. The first time a segment is referenced, the Master Control Program reserves the specified amount of memory, inserts the core address in the descriptor, sets the presence bit to 0, and transfers the segment to core. Any subsequent operation on that segment will find the segment already in memory. If the memory is not large enough to accommodate all the program segments simultaneously, segments will be overlaid automatically. Should a particular segment be overlaid and then required again by the program, it will be loaded again, perhaps to a different location. The only delay which results is that required to transfer the segment from drum to core. Over-laying occurs, however, only when the object program is longer than the number of locations in the core memory: for example, when an 8000-word program is processed in a 4000-word memory.

Subroutines are loaded from a tape library as required. If space permits they are retained in core memory throughout processing; automatic overlays are performed, however, in the same way described for segments.

**Operands.** Numeric quantities may be stored in the Program Reference Table as operands. These words may contain input data, integer program constants, variables, and array entries. The format of an operand is shown in Figure 3-9.

The first three fields of an operand are each one bit in length and contain the identification, the sign of

FLAG	IDENTIFICATION	PRESENCE BIT	DRUM NUMBER		SIZE FIELD (NUMBER OF WORDS IN SEGMENT)	DRUM ADDRESS	CORE BASE ADDRESS
------	----------------	--------------	-------------	--	---	--------------	-------------------

Figure 3-8. Program Descriptor

the mantissa or number, and the sign of the exponent. The exponent field is six bits in length and the mantissa may use as many as 39 bits. An integer value less than  $8^{13}$  (approx.  $5.5 \times 10^{11}$ ) can be recorded. In the case of floating point numbers, the equivalent number of 11.7 significant decimal places can be accommodated. Values may range from  $10^{-46}$  to  $10^{69}$ . Internal representation of integers is consistent with representation of floating point numbers. An integer appears in the system as an un-normalized floating point number. For this reason, it is possible to perform arithmetic operations, comparisons, etc., between integers and floating point numbers without conversion.

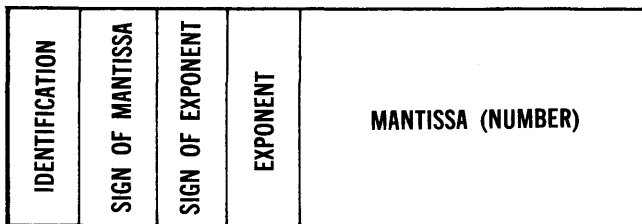


Figure 3-9. Operand

**Operation of Program Syllables in Arithmetic Mode.** In the machine language program of the B 5000, the program word is 48 bits long and is composed of four 12-bit sections called syllables. Figure 3-10 shows the format of a program word. Of the twelve bits in each syllable, two indicate its type and the other ten may contain a numeric value, an address in the Program Reference Table, and an operator such as an add or subtract operator.

There are four types of syllables: literal, operand-call, descriptor-call, and operator. A literal syllable contains a numeric value in the ten low-order bits, which is stored directly in the Stack for use as an operand. Operand-call and descriptor-call syllables contain an address in the Program Reference Table

*call by value*

by which information in storage may be located. An operator contains the code for a particular arithmetic, logical, or other operation to be performed on items which have been placed in the Stack.

These syllables, if described in terms of a conventional machine, may be considered as the instruction words of the B 5000 System. They are referred to as syllables, however, not as instructions.

A program word is brought to the Program register (P register) for processing. The function of this register (see Figure 3-11) is to hold the four syllables of the word currently being executed. Note that each syllable position of the register has a unique address. The syllables are executed in sequence, starting with the left-most syllable. The Control Counter (C register) holds the address of the next syllable to be executed, as shown in Figure 3-12.

**LITERAL SYLLABLE.** This syllable causes the positive integer contained in its ten low-order bit positions to be placed in the Stack. Consequently, it is not necessary to store a constant in the Program Reference Table and reference it with an operand-call syllable. All literal syllables are constructed during compilation and inserted in the program string.

In Figure 3-13 the contents of the literal (the integer 123) are placed at the top of the Stack.

**OPERAND-CALL SYLLABLE.** Execution of this type of syllable causes an operand to be transferred to the Stack. The operand may be obtained by three methods. When an operand-call syllable references the Program Reference Table it encounters one of the four types of information stored there: an operand; a data descriptor; an input/output descriptor; or a program descriptor. The method by which the operand is obtained and placed in the Stack depends upon which type of word is referenced. Each of these cases is described.

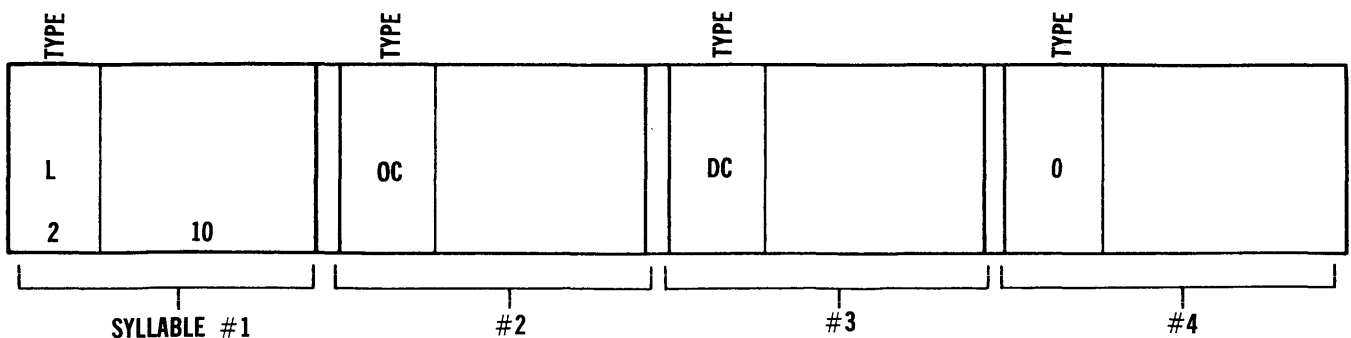


Figure 3-10. Program Word

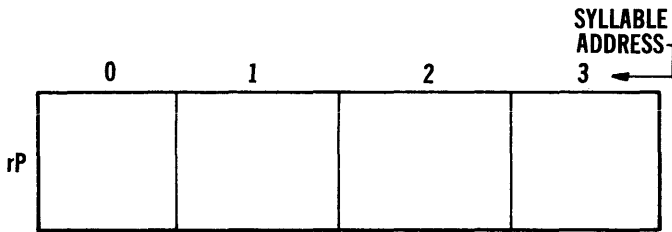


Figure 3-11. Program Register

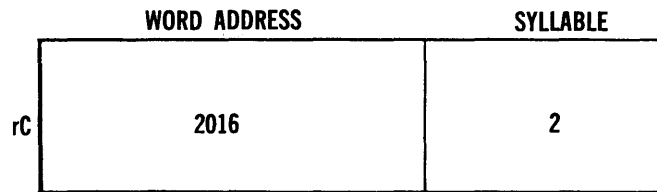


Figure 3-12. Control Register

1. Reference to an operand. When an operand-call syllable encounters an operand in the Program Reference Table, the operand is placed directly at the top of the Stack, that is, in the A register. This procedure is shown in Figure 3-14.

The operand-call referenced location 200 of the Program Reference Table. The R register contains the base address of the table, and so the

referenced location is incremented by the amount 1300. The value located in cell 1500 (integer 77) is placed in the A register, and prior entries are pushed down in the Stack.

2. Reference to a data or input/output descriptor. A process known as indexing occurs any time an operand-call or descriptor-call syllable encounters a descriptor with a size field greater than 0 in the Program Reference Table. The address part of the referenced descriptor (transferred to the A register at stage 1) is automatically incremented by the value held in the B register. The new address is checked to determine if it is within the area defined by the descriptor. The contents of the incremented address are placed in the B register for subsequent use (stage 2).

Two types of indexing are possible. Constant indexing is used when a particular word of a data array is desired. Variable indexing is used when the program requires successive elements in an array; for example, a program may process  $D_I$ , where I changes.

a. Constant indexing. Assume that the fifth word of record S ( $S_5$ ) is to be added to the seventh word of record T ( $T_7$ ). The result is

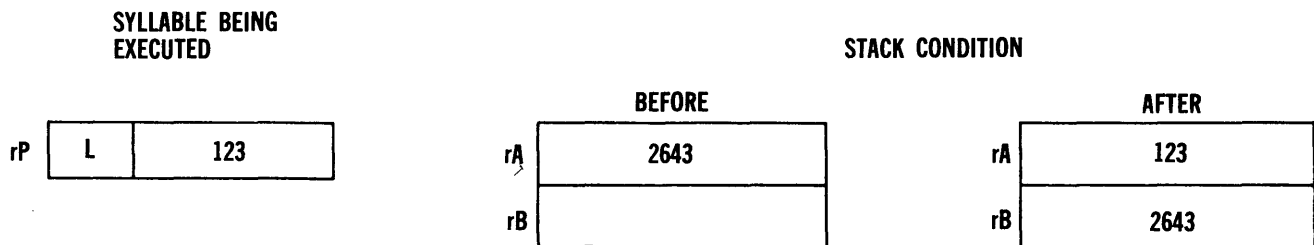


Figure 3-13. Literal Placed on Top of Stack

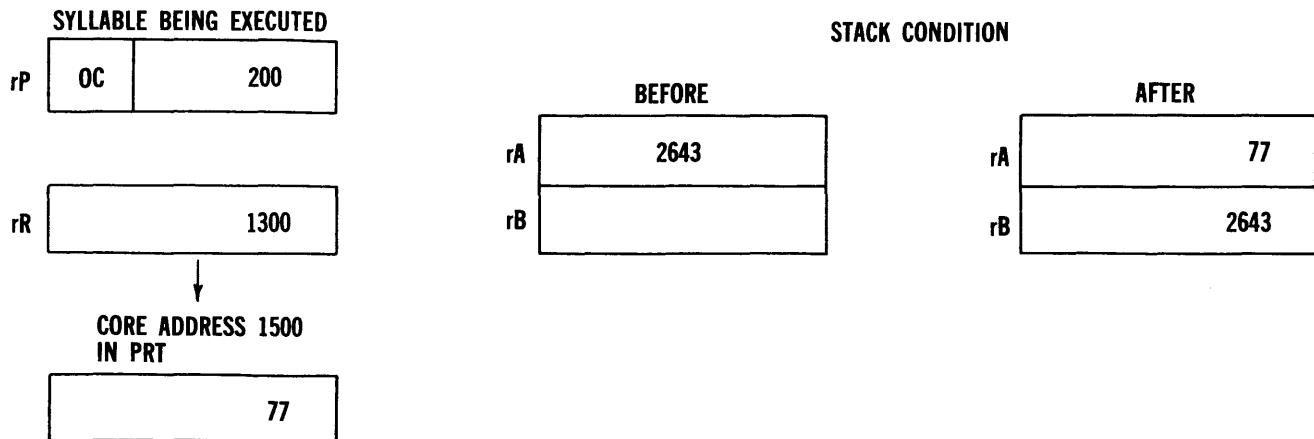


Figure 3-14. Obtaining an Operand Directly From the Program

$S_5 + T_7$   $X_{10}$  Starting addresses for data arrays are:

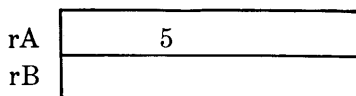
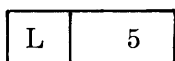
S — core address 200

T — core address 750

X — core address 1000

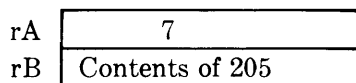
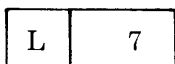
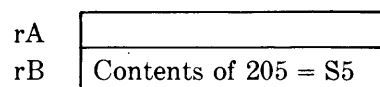
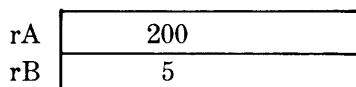
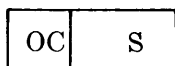
*Syllable Being Executed*

*Condition of Stack*



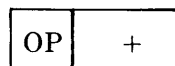
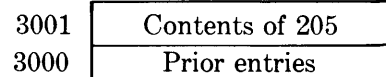
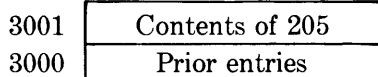
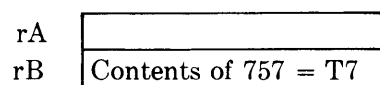
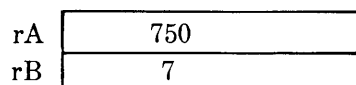
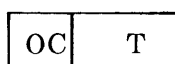
Stage 1

Stage 2—after indexing

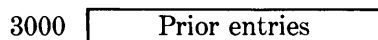
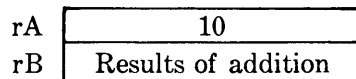
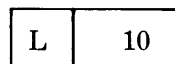
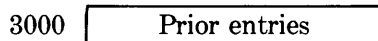
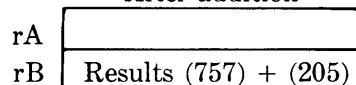


Stage 1

Stage 2

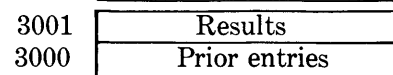
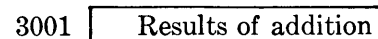
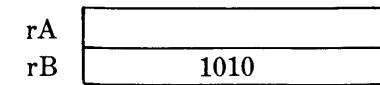
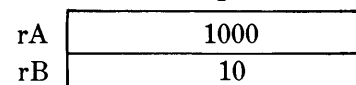
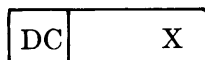


After addition



Stage 1

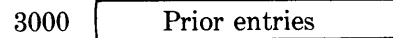
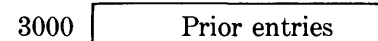
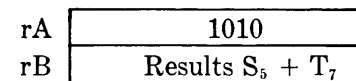
Stage 2



The descriptor-call syllable indexed the address of array X, leaving the incremented address in the B register.

Stage 1

Stage 2



This operator stores the contents of the B register (second Stack location) in the address held in the A register (first Stack location). Therefore the items in the Stack are automatically pushed up so that the address of array X — location 1010 — is in the A register and the results of  $S_5 + T_7$  are in the B register.

Figure 3-15. Constant Indexing



to be stored in the tenth word of record X ( $X_{10}$ ). A literal is used to transfer the desired index to the Stack. Then an operand-call syllable references a data descriptor in the Program Reference Table. The value of the literal is automatically added to the core address of the descriptor. The *contents* of the indexed descriptor address are then placed in the B register. Figure 3-15 shows the effect of the operations  $S_8 + T_7 \rightarrow X_{10}$  on the Stack.

- b. Variable indexing. A similar process occurs when the index is a variable. Any algebraic computation can be performed to get an indexed value. Multi-level indexing can also be performed. For example, in processing the element I of array D, an operand-call syllable is used to transfer the value of I to the A register.

Then, an operand-call syllable brings the data descriptor for array D to the Stack, increments the core address by the value of I, and places the *contents* (since an operand was called for) of the indexed address in the B register for subsequent use. If one wishes to store the element  $D_I$ , an operand-call syllable transfers the value of I to the Stack. Then a descriptor-call syllable brings the starting address of D to the Stack, increments it by the value of I, leaving in the B register the indexed *address*. The store operator, which would follow, will store the contents of the second Stack location in the address specified by the first Stack location.

Refer to Figure 3-16 for a diagram of the Stack operation.

If multi-level indexing is to occur, the same series of operations can be repeated indefinitely. For example, processing the element  $X_{YZ}$  entails the following steps. An operand-call brings the value of Z to the Stack. A second operand-call transfers Y, which is a data descriptor. The address of Y is incremented by Z, and the contents of the incremented address are brought to the Stack. The next operand-call transfers X (which will be a data descriptor), increments its core address by the indexed value already in the B register, and then places the contents of that address in the B register for subsequent use. Figure 3-17 shows this sequence.

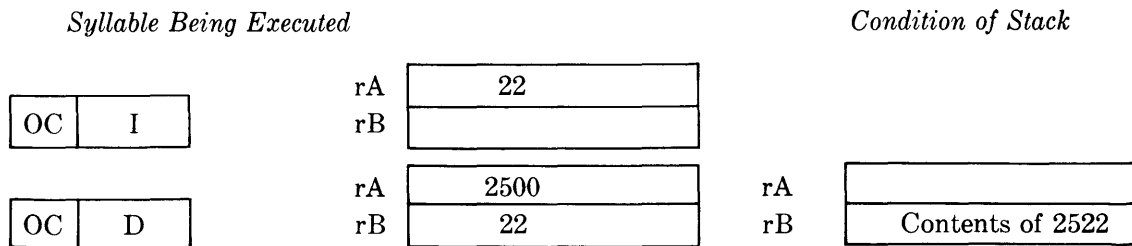
If the presence bit indicates the data is not stored in memory an interrupt occurs and the Master Control Program loads the desired information. If the incremented address is larger than the area defined by the descriptor, an interrupt occurs. The handling of interrupt conditions is discussed in Section 4.

- 3. Reference to a program descriptor. Encountering a program descriptor causes it to be placed at the top of the Stack. Its core address is transferred to the C register, becoming the next syllable to be executed. This is the address of the first word of a program segment, and a transfer of control is executed. Entry to the Subroutine mode is described later in this section.

$D_I$

Array D has a starting address of 2500.

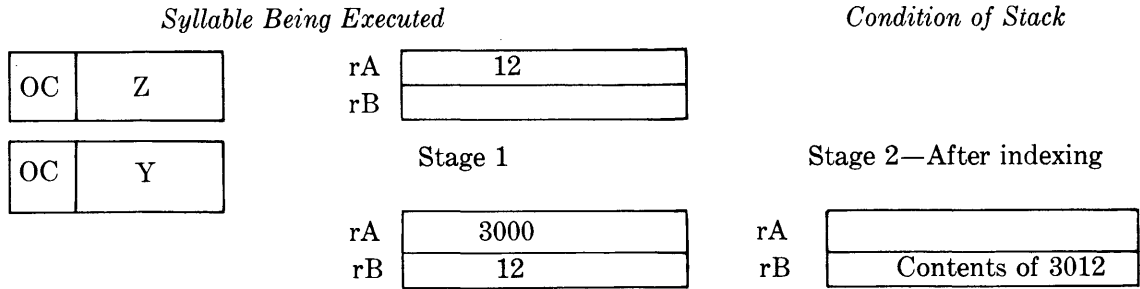
Operand I has a current value of 22; this may be a computed value.



The B register contains the *contents* of location 2522 after indexing.

Figure 3-16. Variable Indexing

$X_{YZ}$   
 Z has a value of 12.  
 Base address of Y is 3000.  
 Base address of X is 3500.



Assume that location 3012 contains the value 16. Now the address of array X—location 3500—is brought to the A register and incremented by the value 16. Finally the *contents* of location 3516 are placed in the B register for use by the program.

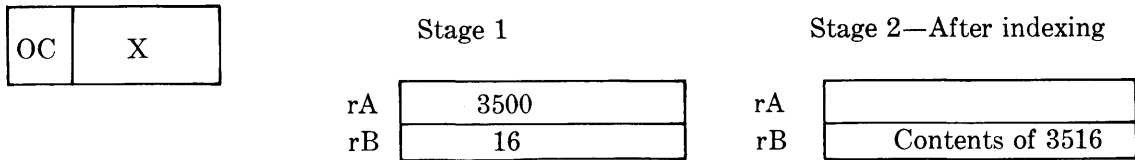


Figure 3-17. Multi-level Variable Indexing

**DESCRIPTOR-CALL SYLLABLE.** This type of syllable causes an address to be placed in the Stack. When a descriptor-call references the Program Reference Table it may encounter an operand; a data or input/output descriptor; or a program descriptor. The method by which the address is obtained and placed in the Stack depends upon which type of word is referenced.

1. Reference to an operand. If an operand is encountered, and brought to the A register, it is replaced by a data descriptor which contains its address in the Program Reference Table. The data descriptor remains in the Stack (A

register), with its size field set to zero. Refer to Figure 3-18.

In this case the core address of the descriptor-call syllable is incremented by the contents of the R register (base address of the Program Reference Table), and address 1201 is referenced. The contents of cell 1201 are examined and when it is determined that the word contains an operand, not a descriptor, the address 1201, rather than its contents, is transferred to the A register. This word is identified as a data descriptor which references an operand value 123 in core address 1201.

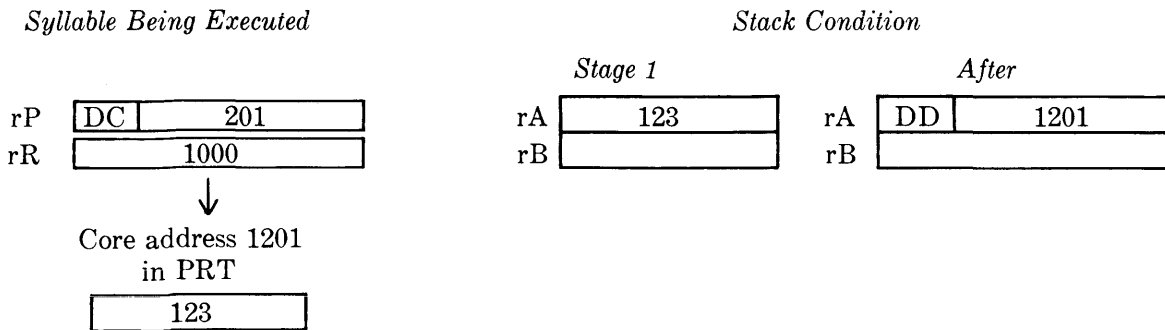


Figure 3-18. Obtaining Descriptor Through Reference to Operand

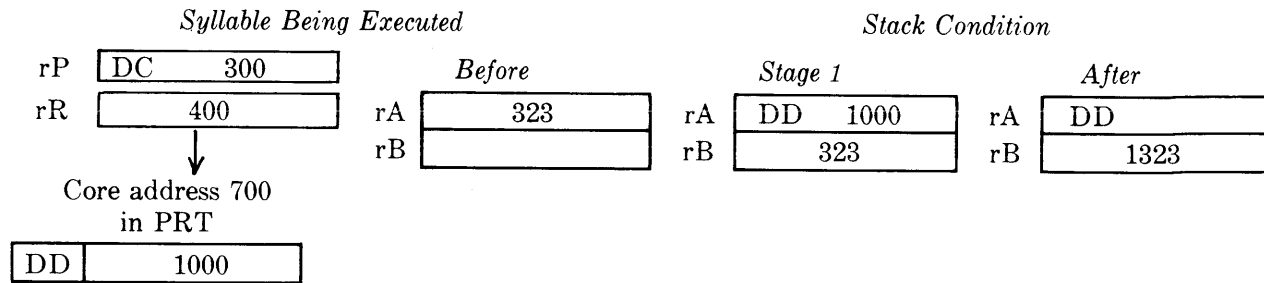


Figure 3-19. Obtaining Descriptor by Referencing Descriptor

2. Reference to a data or input/output descriptor. When a descriptor-call encounters either a data or input/output descriptor, the word is transferred to the A register. The presence bit and size field of the descriptor are checked. The contents of the B register are added to the base address of the descriptor in the A register, and the resulting address is checked to ascertain that the address is within the area defined by the descriptor.

In Figure 3-19 the descriptor-call syllable referenced location 300 of the Program Reference Table. This address is incremented by the base address of the table, and location 700 is examined. The data descriptor encountered in that location is transferred to the A register and incremented by the contents of the B register. The final descriptor references core address 1323.

3. Reference to a program descriptor. The program descriptor is placed in the A register, and its core address is transferred to the C register, becoming the next word to be executed. This is the beginning address of a subroutine; the results of its execution are placed in the A register and control is returned to the main program.

**OPERATOR SYLLABLE.** This syllable type differs from the others in respect to its effect upon the Stack. The other syllables *place* operands and descriptors in the Stack. The operator syllable, in its variations, *manipulates* items in the Stack. The operators used in the Arithmetic mode may be divided into the following categories: unary, binary, and miscellaneous.

Unary operators are associated with only one operand. They affect only the A register and do not delete any items in the Stack. For example, unary operators can alter the sign of the A register by setting it to plus, minus, or by reversing it.

Binary operators require two operands and their

execution deletes one item from the Stack. They operate on values in the A and B registers, producing a result in the B register and leaving the A register empty. If either the A or B registers is empty before a binary operation takes place, it is filled automatically from the top memory location in the Stack.

Arithmetic, logical and relational operators are all binary. The arithmetic operators provide for addition, subtraction, multiplication, and division. Comparisons are made by using relational operators. Logical operators perform such operations as extraction and complementing.

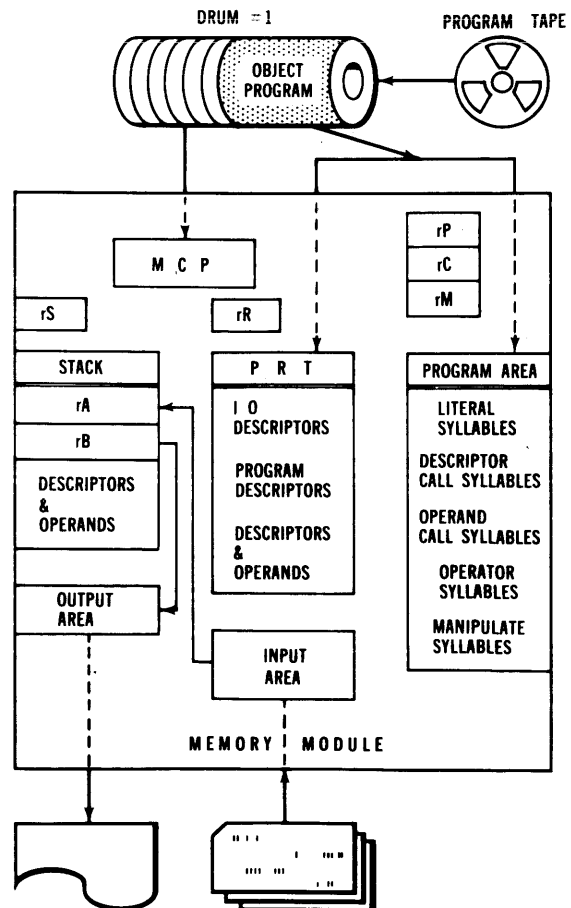


Figure 3-20. Layout of Memory Module With Associated Registers

A group of miscellaneous operators provide for storing the results of arithmetic operations, for conditional or unconditional branching, and for a wide range of control operations.

## SUMMARY

The Arithmetic mode uses three types of words—descriptors, operands, and program words. The descriptors and operands make up the Program Reference Table. This table is an index to all the information required for execution of a program and is also a storage center for numeric quantities. Four types of syllables form program words. The operand-call, descriptor-call, and literal syllables transfer information to the Stack, where operator syllables cause the data to be processed. The registers of the B 5000 system are physically located within the Processor; however, they have been described in connection with the elements of the system which use them. The A, B, and S registers, for example, have been associated with the Stack, the R register with the Program Reference Table, and the P, C, and M registers with the memory storage for program segments. Figure 3-20 is a schematic layout of a Memory Module, showing the interrelation of the major processing components.

## OPERATION DURING SUBROUTINE MODE

### INTRODUCTION

A subroutine can be defined as the repetition of a series of operations during one pass through a program. Control is transferred from the main program to the subroutine and returned upon completion of the subroutine. The structure of the B 5000 programming language extends the usefulness of this type of program organization.

PROCEDURE and FUNCTION declarations in ALGOL 60, and also SECTIONS and PARAGRAPHS in COBOL 61 may be treated as subroutines. Therefore the Subroutine mode of operation in the B 5000 has been designed to assure efficient generalized handling of subroutines to any depth, including recursively defined subroutines.

The Subroutine mode has several characteristics which distinguish it from the Arithmetic mode. Operand-call and descriptor-call syllables can reference parameters and temporary storage in the Stack. These syllables can also reference constants in the subroutine program string. An additional control register, the F register, and two special operators are used with the Subroutine mode. The *mark stack* operator prepares for subroutine entry; the *return*

operator restores the registers to their original condition and transfers control to the main program. These characteristics are discussed in further detail in the following paragraphs, and examples are provided to show the use of the subroutine operators.

## USE OF OPERAND-CALL AND DESCRIPTOR-CALL SYLLABLES

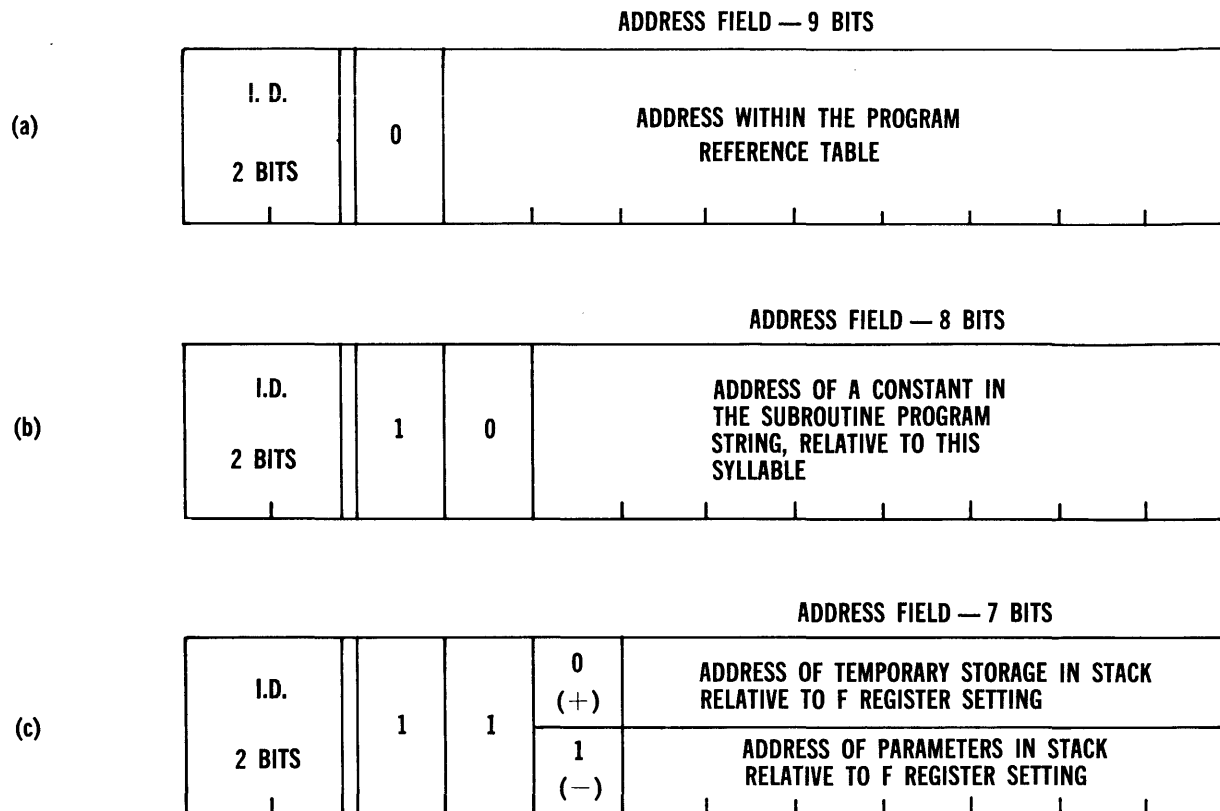
The Subroutine mode is designed so that generalized subroutines, whose parameters are likely to change with each use, may be incorporated with an object program efficiently. The subroutines must be virtually independent of the Program Reference Table, which is specially composed for each object program. To achieve this, the power of the operand-call and descriptor-call syllables is extended in the Subroutine mode so that they can address the Stack directly and use it for input parameters and temporary storage. Constants may be stored in the subroutine string, and these syllables used to address them also.

Normally the parameters required by each subroutine are stored in or referenced by the Program Reference Table. They are transferred to the Stack just prior to the shift from Arithmetic to Subroutine mode. Once control is transferred, the subroutine addresses the Stack for required parameters and its own program string for constants. All the syllable types that are used in the Arithmetic mode are available to the Subroutine mode, however, and under certain circumstances the subroutine may reference items in the Program Reference Table and, through it, data in general storage.

The format of operand-call and descriptor-call syllables in Subroutine mode is illustrated in Figure 3-21. Several indicator bits in the high-order positions of the address field indicate whether reference is to be made to parameters and temporary storage in the Stack, to constants located within the subroutine segment, or to the Program Reference Table.

## THE F REGISTER

The F register records information that links the subroutine being executed to the main program. The information required to return all registers to their condition prior to subroutine entry is stored in the Stack. The address of this Stack location is transferred to the F register and preserved throughout subroutine operation. After the subroutine has been executed, the Stack address contained in the F register is used to locate all the return-point information. When the registers have been restored to Arithmetic mode, control is returned to the main program.



**Figure 3-21. Variations in Format of Operand-Call and Descriptor-Call Syllables in Subroutine Mode**

The Stack address stored in the F register is also used as a base address for referencing subroutine parameters and temporary storage. Parameters, which are transferred to the Stack before control is shifted to Subroutine mode, have addresses that are negative, relative to the F register setting. Temporary storage locations have addresses that are positive, relative to the F register setting. The operand-call and descriptor-call syllable format shown in Figure 3-21 (c) provides for this addressing technique.

Since the C register is used in the Subroutine mode the same way as in the Arithmetic mode (to contain the address of the next program word and syllable to be executed), return to the main program is made to the syllable address restored to it after the subroutine has been completed. It is reasonable for one to ask why the contents of the C register are not merely preserved in the F register. The B 5000 System provides for indefinite nesting of subroutines. Assume that a program calls out a subroutine, and that this subroutine, in turn, requires a second subroutine. If the contents of the C register are stored in the F register upon entry to the first subroutine, they will be lost when the first subroutine stores its

re-entry point in the F register upon transferring control to the second subroutine. Therefore, more complete return-point information must be provided. The information that is stored in the Stack includes the contents of the F register as well as the contents of the C register. As control is returned from each subroutine, then, the F register is restored to its former status. Thus it is possible to have an indefinite nesting of subroutines. Information is always preserved, automatically, for restoration of the C and F registers to their previous condition.

### **SPECIAL SUBROUTINE OPERATORS**

The mark stack operator is used in preparation for transfer to Subroutine mode. When it is encountered in a program syllable the contents of the F register are stored in the Stack; and the Stack address placed in the F register. The main program normally causes one or more parameters to be transferred to the Stack for use during subroutine operation. They are stored in the locations immediately succeeding the one indicated by the F register setting. These parameters must be cleared from the Stack when their

usefulness is exhausted. The mark stack operator indicates the beginning address of parameter storage. The second subroutine operator is called the return operator, and is normally the last syllable in the subroutine. It restores the F register to its previous setting, clears the Stack of all parameters and temporary storage locations, and returns control to the main program by restoring the contents of the C register.

### ENTRY TO SUBROUTINE MODE

After the mark stack operator has set up the Stack for storage of parameters, and these have been transferred to Stack locations, entry to the Subroutine mode is made when an operand-call syllable in the Arithmetic mode references a program descriptor. The following series of operations causes transfer of control:

1. The contents of the F and C registers (return-point information) are stored in the Stack, after any items in the A and B registers have been pushed down into the Stack.

SYLLABLE
1. OP MARK STACK

REGISTERS
rC 1000/1
rS 102
rF 102

STACK
rA
rB
102 76
101 PRIOR ENTRY

Old setting of F register is stored in Stack and F register is set to location (102).

2. OC
X IN PRT

rC 1000/2
rS 102
rF 102

rA X
rB
102 76
101 PRIOR ENTRY

Subroutine input parameter (X) is transferred from Program Reference Table to A register.

3. OC
SIN PROG/DESC.

rC 4000/0
rS 104
rF 104

rA SIN PROGRAM DESC.
rB
104 102 1000/3 RETURN POINT
103 X
102 76
101 PRIOR ENTRY

Return-point and F register setting are stored in Stack 104. (X has been stored in 103.) F register is set to 104. Entry is made to subroutine at location in C register. Assume the base address of the SIN program descriptor is 4000.

2. The Stack address is stored in the F register.
3. The core starting address from the program descriptor is placed in the C register, becoming the next syllable to be executed.

Example A illustrates these steps for entry to a subroutine which calculates SIN(X); Example B show an entry to a subroutine which computes NET PAY. Example C shows subroutine operation when information is contained within the subroutine string. Example D illustrates nesting of subroutines; that is, the first subroutine uses a second subroutine. Exit from the Subroutine mode is shown in Example E. Example A: Calling sequence for a subroutine which calculates SIN(X). The function and operation of each of the three syllables required for this is shown below. Upon return to the main program, the result will be located in rA (the A register) and the Stack configuration and register settings will be restored.

Assume that rC = 1000-0, rS = 101, and rF = 76. The diagram of the registers and Stack indicates their respective contents *after* the syllable has been executed.

Example B: Calling sequence for a subroutine that calculates NET PAY. Assume that NET PAY = GROSS PAY-FICA-WITHHOLDING TAX-HOSPITAL INSURANCE. The amounts for GROSS PAY, FICA, WITHHOLDING TAX and HOSPITAL INSURANCE have already been computed and can be obtained through the Program Reference Table. Before entry, register conditions are as follows:

$$rC = 1000 / 0$$

$$\begin{aligned} rS &= 101 \\ rF &= 76 \end{aligned}$$

The syllables required to enter this subroutine consist of a mark stack operator, and operand-call syllables for values of GROSS PAY, FICA, WITHHOLDING TAX, and HOSPITAL INSURANCE.

These operands are the parameters for the subroutine and are transferred successively to the Stack. Finally, an operand-call references the NET PAY program descriptor.

SYLLABLE EXECUTED	REGISTERS
1. OP MARK STACK	rC 1000/1
	rS 102
	rF 102

Mark stack operator stores contents of F register in Stack.

2. OC HOSP. INS.	rC 1001/2
	rS 104
	rF 102

Series of operand-call syllables has transferred parameters for subroutine to successive Stack locations.

3. OC NET PAY	rC 2200/0
	rS 107
	rF 107

Operand-call syllable references the NET PAY program descriptor, causing address of the first word of this subroutine to be placed in the C register, and control to be transferred. Assume the base address in the NET PAY program descriptor is 2200. Return point information is stored in Stack address 107, and that location preserved in F register.

STACK CONDITION
rA
rB
102 76
101 PRIOR ENTRY

rA	VALUE HOSP. INS.
rB	VALUE-W. TAX
104	VALUE-FICA
103	VALUE-GROSS PAY
102	76
101	PRIOR ENTRY

rA	NET PAY PROG. DESC.
rB	
107	1001/2 102
106	VALUE-HOSP. INS.
105	VALUE-W. TAX
104	VALUE-FICA
103	VALUE-GROSS PAY
102	76
101	PRIOR ENTRY

Example C: Assume the SIN(X) is approximated by some polynomial of the form  $(a_0 + a_1 x + a_2 x^2 + \dots)$  and the coefficients are contained within the program segment.

To perform the first multiplication, the value of X is referenced in the Stack by using an operand-call syllable indicating that the Stack is to be referenced

and that the location is one less (negatively relative to the top of the Stack at last entry) than the value in rF.

The coefficient (a) is obtained by another operand-call, this time indicating that the value is located within the program segment area.

SYLLABLE

OC	111 1
----	-------

REGISTERS

rC	SIN/2
rS	104
rF	104

STACK

rA	X
rB	a <sub>0</sub>

Gets X from Stack.

OC

10 10
-------

REGISTERS

rC	SIN/3
rS	105
rF	104

104	102	1000/3
103	X	
102	76	
101	PRIOR ENTRY	

rA	a <sub>1</sub>
rB	X

Gets coefficient from location SIN-10 in program area.

OP

MULTIPLY
----------

REGISTERS

rC	SIN+1/0
rS	105
rF	104

105	a <sub>0</sub>	
104	102	1000/3
103	X	
102	76	

rA	
rB	a <sub>1</sub> X

105	a <sub>0</sub>	
104	102	1000/3
103	X	
102	76	

Example D:  $\text{COS}(X) = [1 - \text{SIN}^2 X]^{1/2}$

Calling sequence for a subroutine which itself calls

out subroutines.

This will illustrate how a descriptor-call syllable following a mark stack operator functions.

SYLLABLE

1. OP MARK STACK
------------------

REGISTERS

rC	700/0
rS	301
rF	301

PRT

2010	X	
2011	✓	PROGRAM
2012	SIN	DES-
2013	COS	CRIPTOR

STACK

rA	
rB	
301	276
300	PRIOR ENTRY

Save contents of the rF (assumed to be 276).



2. 

OC	2010
----	------

rC	700/1
rS	301
rF	301

rA	X
rB	
301	276
300	PRIOR ENTRY

Puts X into the rA.

3. 

DC	2011
----	------

rC	700/2
rS	301
rF	301

rA	√ PROG/DESC
rB	X
301	276
300	PRIOR ENTRY

Puts program descriptor for square-root subroutine into the rA.

4. 

DC	2012
----	------

rC	700/3
rS	302
rF	301

rA	SIN PROG/DESC
rB	√ PROG/DESC
302	X
301	276
300	PRIOR ENTRY

Puts program descriptor for SIN subroutine into the rA.

5. 

OC	2013
----	------

rC	COS S-R
rS	305
rF	305

rA	
rB	
305	301 701/0 (RETURN POINT)
304	SIN PROG/DESC
303	√ PROG/DESC
302	X
301	276

Pushes contents of rA and rB down; sets return-point and loads the rF with this location.  
Enters COS subroutine.

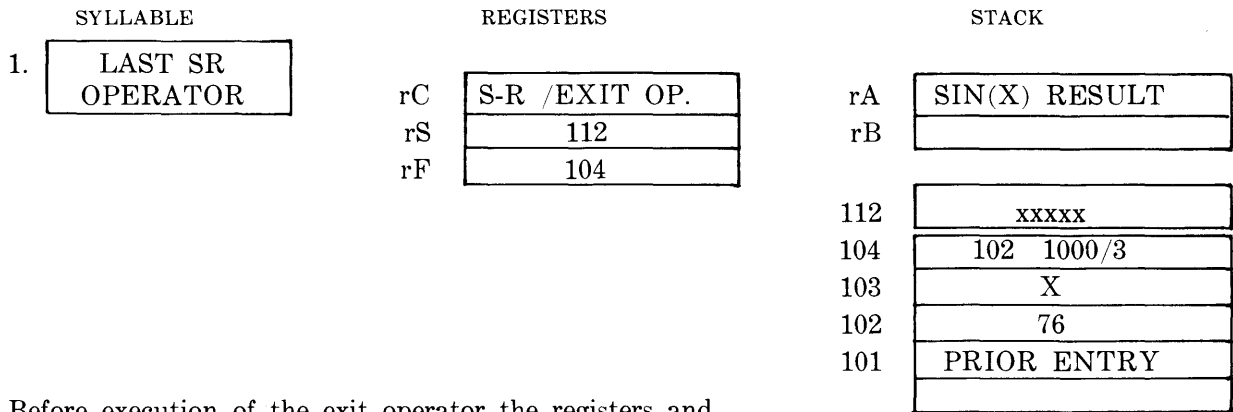
### EXIT FROM SUBROUTINE MODE

The result of subroutine operation is placed in the A register to be used in subsequent operations of the main program.

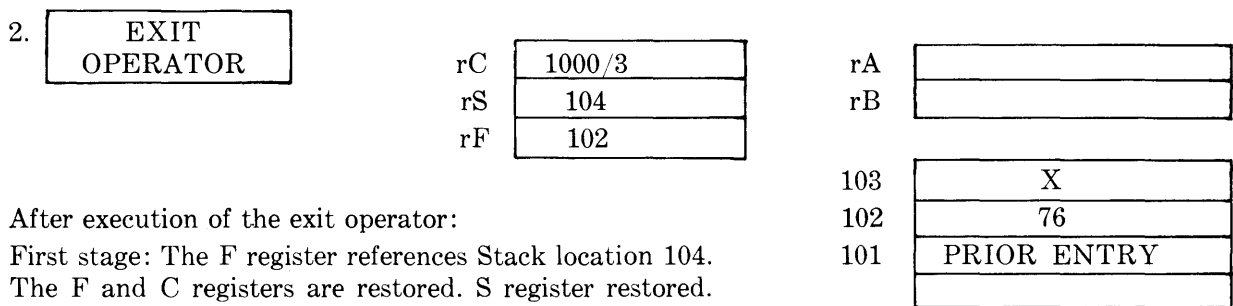
The return operator usually terminates subroutine operation by referencing the F register and, in turn, the Stack address indicated. The first step restores the F register to its setting when the program descriptor was encountered, and replaces the return-point address in the C register. Now the F register

contains the Stack address stored there when the mark stack operator was encountered. This Stack address is referenced: the F and S registers are restored to their condition prior to execution of the mark stack operator. Thus control is returned to the main program and the Stack is cleared of information no longer needed for processing. Example E shows the two stages of exit from the subroutine which calculates SIN(X).

Example E: Exit sequence from the subroutine which calculates SIN(X). Refer to Example A.

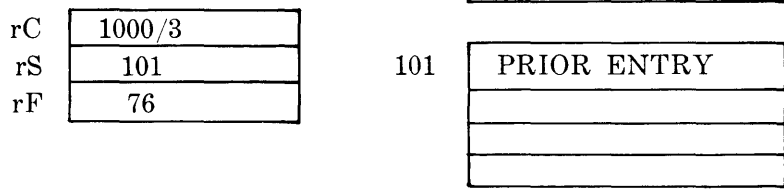


Before execution of the exit operator the registers and Stack have the contents shown.



After execution of the exit operator:  
 First stage: The F register references Stack location 104. The F and C registers are restored. S register restored.

Second stage: The F register references Stack location 102. The F and S registers are restored to settings prior to execution of mark stack operator, as shown.



## OPERATION DURING DATA-MANIPULATION MODE INTRODUCTION

Editing of data in earlier computers was possible only with costly and time-consuming methods. Plug-board wiring, for instance, or elaborate format-control specifications for an input/output device are difficult to prepare and have limited flexibility. Intricate programming results in a loss of time and memory space.

Manipulation and comparison functions generally involve a series of add, shift, mask, and store operations. Even such a simple function as comparison of split-field keys requires much subtracting, shifting, and branching. Thus, comprehensive, flexible com-

parison and data-manipulation abilities have been sacrificed to speed or programming considerations. In recognition of these deficiencies, the B 5000 has been provided with facilities for concise, efficient data manipulation. The Data-Manipulation mode uses individual characters as its basic unit of information instead of the normal word length used in the Arithmetic and Subroutine modes.

Input and output formats, and editing, comparison, and conversion of data are among the most difficult areas of programming. COBOL provides automatic specification for these functions, and in ALGOL they are described by language extensions. Since COBOL and ALGOL are the programming languages of the B 5000, and the "hardware logic" itself includes single-character operation, the B 5000 is able to meet the most stringent editing requirements with ease.

## EDITING FUNCTIONS

The vocabulary of the Processor operating in the Data-Manipulation mode includes a comprehensive set of special operators (single-character instructions). Operators may be used individually or in groups. Combinations of the operators, referred to as editing or manipulation strings, perform the following functions:

1. Move individual six-bit characters, groups of characters, or complete words (eight alphanumeric characters per word) from one area to another.
2. Delete characters or fields of characters from a source area as they are being relocated.
3. Insert characters or fields of characters directly from the manipulation (program) string itself into the edited fields.
4. Provide for overlays or insertions of characters from one area (source area) into a second (destination) area.
5. Provide a complete set of comparison operators to permit comparison of source- and destination-area characters or fields, or tests against characters from the manipulation string itself.
6. Provide unconditional jump (change of execution sequence) operators to direct control within the manipulation string, and a complete set of conditional jump operators to interrogate the results of the comparison and test operators and direct control accordingly.
7. Provide a set of repeat operators which allow segments of the manipulation string to be repeated. These repeat operators may be combined (nested), thus permitting complete flexibility in the execution of the manipulation string.
8. Provide a set of operators to permit alteration of the normal execution sequence out of "nests" or loops as a result of a relational operator comparison.
9. Provide operators to permit interchange of source and destination areas. By interchanging the contents of the associated character-location registers, interspersing or merging characters from the two areas is possible.
10. Provide a set of operators to permit manipulation, retention, and recovery of the character location registers associated with the source and destination areas. These operators are particularly useful in sorting and character-recognition applications.

11. Provide operators to permit transfer of only the "zone" or only the numeric portion of characters or fields, and automatic recognition of the transferred field's algebraic sign. These operators are used primarily in handling characters represented in standard punched-card notation.
12. Provide operators to permit automatic decimal-to-octal and octal-to-decimal conversion. By utilizing these operators, it is no longer necessary for the programmer to provide the coding or the conversion algorithms to the Processor.
13. Provide operators to communicate with the Normal mode. These operators are used to transmit specifications or parameters from the Normal mode to the Data-Manipulation mode so that general-purpose data-manipulation procedures or functions can be used. This is done in much the same way parameters are supplied for subroutine operation.
14. Provide decimal addition and subtraction operators.
15. Provide a complete set of individual binary-bit manipulation operators.

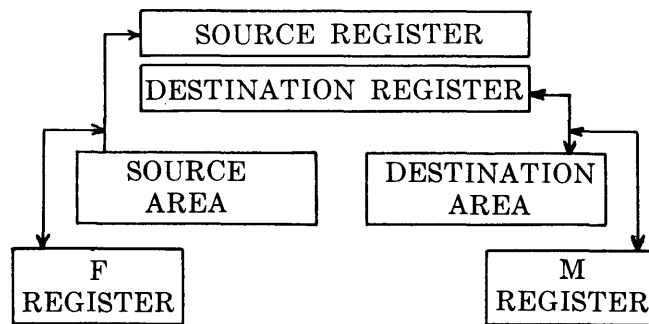
## ENTRY TO DATA-MANIPULATION MODE

The normal arithmetic and control registers of the Processor are used to provide the necessary hardware functions for the Data-Manipulation mode. Entry to and exit from the Data-Manipulation mode is accomplished by providing the Processor with the ability to shift modes in much the same manner as in subroutine handling. The enter-data-manipulation-mode operator (instruction syllable), encountered during the normal syllable execution of a program string or sequence, signals the Processor to store all necessary registers and information for return to the Normal mode. The Processor then recognizes three descriptors which supply (a) the location of the "source" information or information to be manipulated or handled; (b) the location of the "destination" area or fields for comparison, movement, or editing; and (c) the location of the manipulation instructions or program string to be employed.

Words from one area (source) are loaded into the top register of the Stack (A register), and assembled into, or operated against, characters from a second area (destination) in the second register of the Stack (B register). The Subroutine register (F) maintains control over the memory location of source characters. The M register controls communication with the destination area.

Transfer of characters to and from the source (A) and destination (B) registers is automatic. For example, during the comparison of 63 characters in the source area against 63 characters in the destination area, the A and B registers are filled eight times. As soon as comparison of the first eight characters is completed, the registers are automatically reloaded with the second group of eight characters, etc. Since the source and destination registers are controlled independently of each other, characters may be manipulated regardless of their position within the memory words. In the comparison operation described above, for instance, each of the 63-character fields could begin with a different character position within the word.

The source-address register and a destination-address register control movement of characters through the Stack levels as illustrated in this flow diagram below:



Although the programmer writes his data-manipulation functions in problem language (ALGOL or COBOL), several of the individual operators, or Processor "instructions" used by the compilers, are explained below. The 12-bit syllable operators utilized by the Processor in performing the specified operations are illustrated. These operators are made up of a six-bit count, specifying the number of times the operation is to be performed, and a six-bit alphanumeric character which specifies the operation to be performed.

Since six bits are allocated for the count of a given operator syllable, decimal 63 (octal 77) is the maximum count possible with one operator. Operators may be repeated, however, to achieve any count desired. As in the Normal mode, each computer word in the manipulation string contains four syllables.

### ILLUSTRATION OF SOME SPECIFIC OPERATORS USED BY THE COMPILER

COUNT	OPERATOR CHARACTER	OPERATOR FUNCTION
nn	A	Transfer nn characters from source to destination area. For instance, <b>23 iA</b> transfers the next 23 characters from the source to the destination area. This syllable performs the operation specified by the MOVE verb in COBOL.
nn	/	Rapid transfer nn complete words (eight characters per word) from source to destination. Words must be aligned to their first character positions. Automatic alignment to the first character of the next word in source or destination field, if they are not aligned, is provided. For instance, <b>10 /</b> transfers 10 complete eight-character words from source to destination area.
nn	L	Insert the following nn characters from manipulation string into destination area. If nn is odd, the first character position will be ignored (lower case b indicates a blank) to maintain normal syllable handling. For instance, <b>06 L</b> <b>b T O T A L 13 A</b> inserts the word "total" into destination area, followed by the next 13 characters from the source area.

COUNT	OPERATOR CHARACTER	OPERATOR FUNCTION
nn	S	Skip over nn characters in destination area, leaving skipped characters unchanged. This operator permits overlays and insertions of characters in destination area. <u>01:S</u> <u>05:A</u> <u>01:S</u> <u>14:A</u> This one-word string would produce, in the destination area, one character unchanged; five characters inserted from the source area; one character unchanged; etc.
nn	D	Skip forward nn consecutive characters from source area upon transfer to destination. <u>06:D</u> <u>02:A</u> <u>06:D</u> <u>02:A</u> These syllables would skip six characters; transfer two; skip six characters; etc., upon transfer of information from source to destination.
nn	operator	Compare nn consecutive characters from the source area against nn consecutive characters from the destination area. The true/false indicator is set to true if the comparison operator's conditions are satisfied, otherwise set to false. The individual operator's test conditions are as follows: <u>nn:C</u> tests "greater than" condition; <u>nn:6</u> tests "greater than or equal to"; <u>nn:7</u> tests "equal to" condition; <u>nn:8</u> tests "less than or equal to"; <u>nn:9</u> tests "less than" condition; and <u>nn:0</u> tests "not equal to" condition. <u>04:S</u> <u>12:0</u> would skip four characters in the destination area and test the next 12 characters from the source area against the next 12 from the destination for "not equal to" and set the true/false toggle as a result.
ch.	op.	Test any specified alphanumeric character (ch.) from the editing string against the next available character from the source area. The operator does not move a character from the source area. The testing operators are as follows: <u>ch.:T</u> tests "greater than" condition; <u>ch.:1</u> tests "greater than or equal to" condition; <u>ch.:2</u> tests "equal to" condition; <u>ch.:3</u> tests "less than" or "equal to" condition; <u>ch.:4</u> tests "less than"; <u>ch.:5</u> tests "not equal to" condition. The true/false indicator is turned on if the comparison is true and off if it is false. The characters specified in the testing syllable may be any of the valid characters as described in the Burroughs Common Language. The syllable <u>0:5</u> , when executed, would turn on the true/false toggle if the next available characters from the source area were not a zero.
nn	(...)	This pair of operators directs the Processor to repeat a segment of the editing string between the left and right parenthesis nn times. These repeat pairs may be nested (repeat pairs within repeat pairs) to any depth. A 00 count on the left parenthesis operator will cause skipping of the manipulator string to the matching right parenthesis operator. Note: In the following examples, the character b will indicate "blank." <u>02:(</u> <u>05:L</u> <u>b</u> <u>T</u> <u>O</u> <u>T</u> <u>A</u> <u>L</u> <u>05:(</u> <u>01:L</u> <u>b</u> <u>\$</u> <u>05:A</u> <u>01:L</u> <u>b</u> <u>.</u> <u>02:A</u> <u>00:)</u> <u>00:)</u> This editing string of three and one-half computer words would produce, in the destination area, two groups of data. Each of the two groups would consist of the word "total" followed by five fields of information from the source area. Each field would be made up of a dollar sign, inserted from the editing string; five characters from the source area; a decimal point, inserted from the editing string; and two characters from the source area.
00	→	This operator directs the Processor to jump out of a repeat nest if the result of a relational test is true. This operator takes precedence over the iteration control exercised by the parenthesis operators and sets iteration count to zero. <u>10:(</u> <u>01:(</u> <u>05:7</u> <u>00:→</u> <u>02:L</u> <u>X</u> <u>X</u> <u>00:)</u> <u>00:)</u> This editing string would compare ten 5-character groups from the source field with ten groups in the destination field. If comparison is unequal, the characters XX are inserted as flags after each comparison.
nn	J	Unconditional jump forward nn syllables in the editing string.

COUNT	OPERATOR CHARACTER	OPERATOR FUNCTION
nn	K	Unconditional jump backward nn syllables in the editing string.
nn	N	Jump forward nn syllables if the result of a relational operation was true. <code>60:(b:2:02:N:01:A:01:J:01:D:00:)</code> This example would test 60 consecutive characters from the source area; delete all blanks; and transfer all other characters to the destination area. Such a string may be used to pack characters into the destination area.
nn	U	Jump backward nn syllables if the result of a relational operation was true.
nn	X	Reverse skip character positions in the source area <code>50:(08:2:00:→:08: X:00:)</code> This string will perform an equality search of a table of 50 entries, each entry being eight characters in length, in the destination area against an eight-character key in the source area. If any of the entries in the table are equal to the key, set the true/false toggle "on" upon exit from the nest.
00	M	Return to Normal mode of operation. All registers will be reset from the Stack to their original contents prior to entering data-manipulation mode.
nn	Y	Reverse skip nn character positions in the destination area.
00	I	Interchange the contents of the F and M registers. This has the effect of reversing the information flow to permit transfer of characters from destination to source area. Associated Stack registers are automatically loaded from new setting after control registers have been interchanged.
nn	F	Set count register to nn. Operator thus provides flexibility of iteration execution.
nn	G	Increase count operator by nn.
n	#	Transfer only zone bits (A and B bits) of nn characters from source to destination area. This operator would normally be used to allow manipulation of algebraic sign or control bits in standard punched-card representation. Numeric bits of receiving field remain unchanged.
nn	\$	Replaces numeric portion (1-2-4-8 bits) of nn characters in destination area by numeric part of corresponding characters from source area.  If the last character transferred contained a minus indication, the true/false indicator is turned on. Zone bits of receiving field remain unchanged.
nn	.	Store source-character location-count register (F) into stack as parameter number nn. The operator thus permits retention of a position in a table or field.
nn	*	Store destination-area character-location register (M) in Stack as parameter number nn. After such a store, the contents of register M may be altered as desired and later recovered with the following two operators.
nn	@	Reset source-area character-location register (F) from Stack with parameter number nn. Operator is used to recover position count stored in stack by one of the two store-parameter operators noted above.
nn	□	Reset destination-area character-location register (M) from Stack with parameter number nn. Typical use of these store and recovery operators in actual syllable strings is illustrated in Example No. 1 following the editing operator list.
nn	E	Obtain numbered parameter from stack and place it in the repeat count register (T). This operator would permit the number of iterations of an editing segment to be specified by the Normal mode.
nn	H	Place register (X) into the stack in parameter position nn. In the execution of a nest of syllables containing a conditional exit operator the actual number of loops or iterations could be determined.

COUNT	OPERATOR CHARACTER	OPERATOR FUNCTION
nn	V	Convert nn alphanumeric decimal characters to octal representation. If the value of nn is less than 11, integer representation will be right-justified in the resulting word. If nn is 12 or more, overflow condition may be produced and the most significant octal characters are lost.
nn	B	Convert a word in octal representation to nn (a maximum of eight) alphanumeric decimal characters. Overflow will be produced if the octal word produces more than specified number of decimal characters.
nn	^	Decimal-add a field of nn characters from source area to nn characters in destination field. Result will be stored in destination area. Overflow may be produced. Algebraic signs are taken from zones of low-order character of each field.
nn	v	Decimal-subtract a field of nn characters in source area from nn characters in destination area. Result will be in destination area. Overflow may be produced.

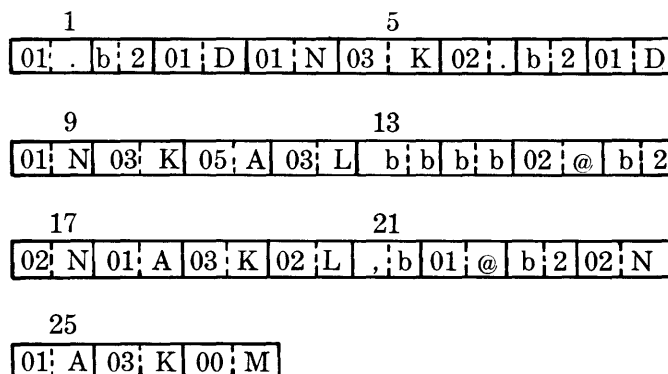
In addition to the operators noted above, there is a complete set of operators for individual bit manipulation to facilitate generation of logical operators, descriptors, etc.

### DATA-MANIPULATION MODE — EXAMPLE 1

**Given:** A field in the source area consisting of employee name and number, each separated by blanks with the following format: First name; blank; last name; blank; five-digit employee number. For example: JOHN b CROY b 15347.

**Object:** Transfer record to destination area, transposing to the following format: Employee number; three blanks; last name; comma and one blank; first name. For example: 15347 bbb CROY, b JOHN.

**Manipulation String:**



Syllable  
Number

Explanation

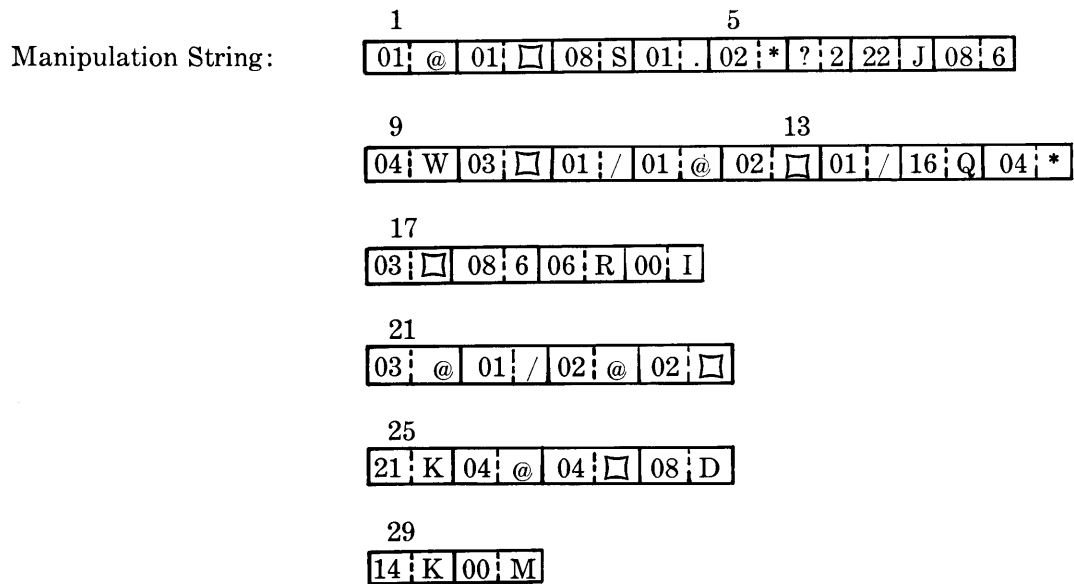
- |    |   |
|----|---|
| 1  | Save current source-character location in parameter 1.  |
| 2  | Test source character for blank.  |
| 3  | Skip to next source character.  |
| 4  | Jump to syllable 6 if relation was true.  |
| 5  | Unconditional jump back to syllable 2.  |
| 6  | Save source-character location in parameter 2.  |
| 7  | Test next source character for a blank.   |
| 8  | Skip to next source character.  |
| 9  | Jump to syllable 11 if relational test was true.  |
| 10 | Unconditional jump back to syllable 7.  |
| 11 | Transfer the next five characters from the source to the destination area.  |
| 12 | Insert three blank characters from the editing string.  |
| 15 | Set source-character location register (F) from parameter 2 in the stack. Parameter was placed there by syllable 6. |

16	Test next source character for a blank.
17	Jump to syllable 20 if result of relational test was true.
18	Transfer one character from source to destination area.
19	Unconditional jump back to syllable 16.
20	Insert a blank and a comma into the destination area from the editing string.
22	Set source character-location register from parameter 1. This parameter was placed into the stack by syllable 1.
23	Test next source character for a blank.
24	Jump to syllable 27 if the result of the previous relational test was true.
25	Transfer one character from source to destination area.
26	Unconditional jump back to syllable 23.
27	Return to Normal mode of operation.

**DATA-MANIPULATION MODE — EXAMPLE 2**

Given: A block of single-word records having an 8-character key. Block will be preceded by a word containing all blanks and will be followed by a word containing all question marks (?). Upon entry to manipulation syllable string, stack parameter 1 will contain the starting location of the block, and parameter 3 will contain the location of a one-word temporary storage.

Object: Sort block of information into ascending sequence using the shuttle exchange method.



Syllable Number	Explanation
1	Set source-character register from parameter 1.
2	Set destination-character register from parameter 1.
3	Skip destination-location register to second word location.
4	Store source-address register into parameter 1.
5	Store destination register into parameter 2.
6	Test first character of the following word to determine end-of-block and, thus, end of sort.
7	Jump to return syllable on true condition.
8	Test source word against destination word for "greater than or equal to."
9	Jump back to syllable 4 on false condition.
10	Set destination-address register to location of temporary storage.
11	Rapid transfer lower-value word from block to temporary storage area.



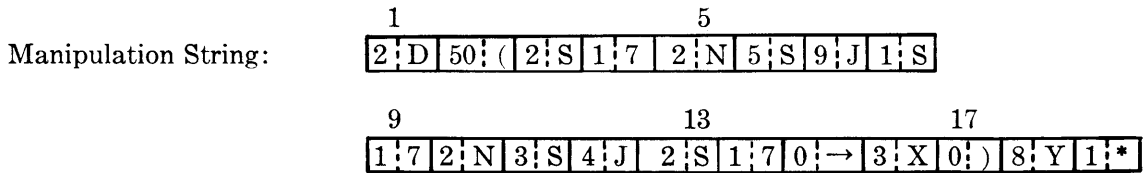
- 12 Set source-address register from parameter 1.
- 13 Set destination-address register from parameter 2.
- 14 Move one word position up in table.
- 15 Reverse both source- and destination-address registers two words.
- 16 Store destination-address register into parameter 4.
- 17 Set destination-address register to location of temporary storage.
- 18 Test source word against destination word for "greater than or equal to."
- 19 Jump forward to syllable 26 on false condition.
- 20 Interchange source- and destination-address registers.
- 21 Set source-address register from parameter 3.
- 22 Insert current low-value word from temporary storage into its relative position in sorted sequence.
- 23 Set source address from parameter 2. Parameter contained location of low-value word in original unsorted sequence.
- 24 Set destination-address register from parameter 2.
- 25 Unconditional jump backward to syllable 3 to continue checking.
- 26 Set source-address register from parameter 4.
- 27 Set destination-address register from parameter 4.
- 28 Skip forward one word in destination-address register.
- 29 Unconditional jump backward to syllable 14.
- 30 Operator to signal return to normal mode of operation when sort is completed.

**DATA-MANIPULATION MODE — EXAMPLE 3**

Given: A search key is the 3-4-5 character position of a word, and a 50-entry table in random numeric sequence, having the corresponding key positions in character positions 3-5-8 of each word entry.

Object: Find word location in the table of the entry having a key equal to the search key. Word location of equal key to be place in stack parameter 1. If search key is not in table, set true/false toggle to false. For example:

Search Key	Table Entries
n n 217 n n n	n n 7 n 3 n n 4
	n n 2 n 8 n n 3
	n n 7 n 1 n n 6
	n n 2 n 1 n n 7
	. . . .



Syllable Number	Explanation
1	Skip over two characters in source (key) word.
2	Repeat string within the following parentheses fifty times.
3	Skip two characters in destination (table) area.
4	Test first key character against first table character.
5	Jump forward two syllables on true condition.
6	Skip remaining characters in table word on false condition.
7	Unconditional jump forward to end of repeat segment.
8	Skip one character in destination word.
9	Test source against destination character for "equal to" conditions.
10	Jump forward two syllables on true condition.

11 Skip remaining characters in destination word on false condition.  
12 Unconditional jump forward to end of repeat string.  
13 Skip two characters in destination area.  
14 Test source against destination for "equal to" conditions.  
15 Jump out of repeat segment on true condition.  
16 Go back to beginning of key word in source location register.  
17 End of repeat nest; go back to syllable 2 to adjust and test iteration count.  
18 Decrease destination-location register upon exit from repeat nest to produce  
location of word containing entries satisfying "equal to" condition.  
19 Store location of last or equal table word into parameter 1.



# SECTION 4

## MASTER CONTROL PROGRAM

### INTRODUCTION

The Master Control Program consists of a special set of routines designed by the manufacturer to provide automatic control over scheduling, allocation of memory, input/output operations, assignment of hardware functions, multi- and parallel processing, and all interruptions of system operation. The Master Control Program (MCP) is permanently recorded on Storage Drum 1 by the manufacturer. Whenever it is being used, the Processor operates in a special mode—the Control mode.

The features and purpose of the B 5000 Master Control Program are explained in this section. Operation during the Control mode and the functions of each of the routines which constitute the Master Control Program are described.

Computers have been developed to save time and to reduce the chance of errors. By providing facilities for performing extremely complex computations and for eliminating tedious clerical functions, they enable users to make creative use of masses of information. Still, operator intervention in computer functions is time consuming and introduces the chance of error. One reason the B 5000 is an advanced system is that it incorporates the means for fully automatic operation. The features of the MCP encompass three general areas: computer functions, automatic system assignment and coordination, and Multi-Processing.

### COMPUTER FUNCTIONS

The MCP controls all computer functions except physical tape and card handling. It provides supervisory control instructions to the operator so that he can make major decisions in directing the processing, but provides the means for automatic handling of all but the most unusual situations. The operator is always able to communicate with the system but seldom needs to. The MCP can analyze error conditions and provide an appropriate course of program action if the object program does not specify one.

### AUTOMATIC SYSTEM ASSIGNMENT AND COORDINATION

Certain operations, which were the responsibility of

the programmer with other computer systems, are performed automatically with the B 5000. By sensing the system's environment, the MCP is able to assign memory areas to program segments, input/output areas, and Program Reference Table entries. Input and output files are automatically identified and given physical unit designations. Standard tape-handling procedures provide for labeling, end-of-tape, and end-of-file conditions.

### MULTI-PROCESSING

The MCP can determine the optimum sequence and combination for processing a batch of jobs. When new or higher priority jobs are introduced, it can adjust its schedule and reorganize the system environment to accommodate the new work. The MCP allocates memory to gain maximum efficiency in processing, and reallocates whenever necessary to preserve optimum processing of a reordered job sequence. The MCP also controls the execution of individual program segments and initiates all input/output operations. Because of its ability to oversee all the programs being processed as well as the individual segments, it can keep input/output devices operating at maximum efficiency with little sacrifice of time.

### ELEMENTS OF THE MASTER CONTROL PROGRAM

The MCP includes a group of special-purpose routines that anticipate and provide for almost all operating circumstances.

1. The Executive Routine coordinates the operation of the MCP by determining what type of control function is required and transferring control to the proper routine. It also supervises input/output operations and the execution of all program segments in memory. When processing is interrupted, it determines the cause. If necessary, the operator is notified; otherwise the Executive Routine takes action either to rectify the situation or to continue processing on another program segment.

2. The Schedule Routine evaluates the priority and equipment requirements of a batch of programs. It schedules these and, if necessary, reschedules to maintain efficient and continuous processing.
3. The Environment Control Routine allocates memory space and input/output devices according to the specifications provided by the Schedule Routine.
4. The Exceptional Condition Routines provide standard error-handling procedures.

### **ENTRY TO THE CONTROL MODE**

The Processor transfers to the Control mode whenever processing is interrupted. Any circumstance that stops processing is called an interrupt condition. It may be caused by operator communication with the system, by developments in the program being executed, or by the hardware.

When an interrupt condition is encountered, control is transferred from the object program to the MCP after execution of the program syllable is complete. The contents of the arithmetic and control registers are stored in Stack locations, and the highest address of these locations is placed in the top of the Stack so that return to the object program can be made. The Executive Routine determines the cause of the interrupt and then transfers control to the appropriate portion of the MCP for handling of the condition. When the interrupt condition has been satisfied, registers are restored and control is transferred to an object program.

### **THE INTERRUPT CONCEPT**

The term "interrupt condition" is used in a special sense with the B 5000 System. It does not imply that work is interrupted or that the system is held up in any way. Rather, a transfer of control is taking place, and the MCP may initiate certain types of operations that can proceed simultaneously with computation. Input/output operations, for example, have usually been controlled by the individual program in process. In the B 5000 System, input/output operations are part of a centralized communications control, and cause an interrupt condition each time they are executed. Since the relatively slow speed of input/output devices normally causes a certain amount of idleness in the Processor, especially for business applications, the B 5000 has been designed to permit a maximum use of all peripheral devices. The minute amount of processing time that is lost through interrupt conditions is far outweighed by the over-all increase in production. Each time an interrupt condition indicates to the MCP that an

Input/Output Channel is free, processing pauses momentarily so that the MCP can initiate new operations to make full use of all the system components. In summary, then, interrupt conditions provide opportunities for the MCP to transfer control to the area of processing which will use the equipment most effectively, or for the MCP to respond to any error signals, operator messages, or unexpected processing conditions. Because it has over-all control of the system, the MCP is able to make the most of the available system and to keep the jobs moving.

In the following pages, the parts of the Master Control Program are examined in detail. The function of each routine and its relationship to other routines, particularly to the Executive Routine, are outlined.

### **EXECUTIVE ROUTINE**

The Executive Routine is loaded initially from the drum when the Program Load button is depressed and is retained in memory throughout processing. It loads other portions of the MCP from the drum as they are needed. As the coordinating member of the MCP, the Executive Routine has six basic functions:

1. To initiate all input/output operations;
2. To analyze all interrupt conditions and provide an appropriate course of action;
3. To maintain control transfer points for program segments;
4. To control the use of other routines in the MCP;
5. To maintain an operations log;
6. To maintain an internal physical system description.

### **INPUT/OUTPUT OPERATIONS**

The Executive Routine, in order to initiate and coordinate all input/output operations, maintains several tables in memory. The base locations of the input/output descriptors for all programs in memory are recorded in one of these tables. A second table keeps a tally of the total number of such descriptors in each Program Reference Table. Constant access to the base location and the total number of each set of descriptors enables the Executive Routine to reference a particular descriptor when an input/output operation is to be executed. A third table contains a record of the descriptors currently being processed by each Input/Output Channel. Thus the Executive Routine can evaluate the status of any descriptor at any time. In summary, the Executive Routine has constantly updated information on the location and status of all input/output descriptors.

### **Initiation of an Input/Output Operation.**

Modification of the status bit in an input/output descriptor causes the operation to be executed. Initially, the status bit of each descriptor is set to indicate that the area referenced is *not* ready for an input or output operation to take place. When an object program has processed all the data in an input area, or filled an output area with information, it executes a program-release operator. This causes the status bit of the corresponding descriptor to be altered, indicating readiness for an input/output operation. When an Input/Output Channel is available, an interrupt condition occurs, allowing the Executive Routine to initiate the operation specified in the altered descriptor. It should be emphasized that processing is interrupted only when a channel is ready and only long enough for the Executive Routine to *initiate* the specified operation, after which the input/output device proceeds independently of the Processor. Control is immediately returned to a program segment.

Consider now the role of the Executive Routine. The object program has indicated readiness for an input/output operation by executing a program-release operator and causing a descriptor to be altered. The Executive Routine locates this descriptor in the appropriate Program Reference Table. This is accomplished by a scanning process: using the base-location table for input/output descriptors and the tally which indicates the total number of input/output descriptors in that table, the Executive Routine examines the status bit of each in turn. A location counter keeps track of the descriptors that have been checked, and when input/output descriptors are referenced in a regular sequence, scanning is reduced to a minimum because it provides a starting point from the last descriptor processed.

Since simultaneous input/output operations can be executed on the B 5000, it is possible that several descriptors from one Program Reference Table may be in an altered condition. Therefore, when the Executive Routine encounters a descriptor with an altered status bit, it checks the list of descriptors currently being processed on each of the Input/Output Channels. If the input/output operation specified by this descriptor has already been initiated, scanning continues until another descriptor with an altered status bit is located or until it is determined that all channels are busy.

**Use of the Continuity Bit.** Programs with a preponderance of input/output operations frequently use multiple read or write areas so that there is always storage for data being read in or for results waiting to be output. Thus processing is not delayed

while necessary data is loaded or while results are cleared from the output area. Multiple read and write areas are normally used in a regular sequence, and the input/output descriptors that reference these areas may be linked by a continuity bit so that each descriptor references the one to be executed next.

The presence of a continuity bit permits the Executive Routine to process a sequence of descriptors without individual program releases. When a program release is executed and a group of linked descriptors is referenced, the status bits of the group are altered automatically in sequence. If Input/Output Channels are available, this alteration permits the Executive Routine to initiate the series of operations without pauses, selecting the read or write areas in turn.

### **Completion of an Input/Output Operation.**

When the specified operation has been executed, the Input/Output Channel causes an interrupt condition by sending an external-result descriptor to the Executive Routine. This descriptor indicates whether or not the operation was completed successfully and provides information regarding the type of operation performed; unit designation; end-of-file or end-of-tape condition; and presence of parity or other errors. Three courses of action are then possible:

1. If the operation was successfully completed, the Input/Output Channel is made available immediately, and the status bit of the input/output descriptor just processed is restored to its original condition. If another descriptor is ready for processing, the Executive Routine initiates the specified operation before transferring control to a program. If no input/output descriptor is available, control is returned either to the segment being processed, or to the next one scheduled in the program backlog table (refer to Schedule Routine).
2. If the external descriptor indicates that a drum-transfer operation has just been completed, the Executive Routine determines whether part of the Master Control Program has been loaded into memory. If so, control is transferred to it.
3. If the operation was not successfully completed, the Executive Routine examines the external descriptor for the cause of failure and attempts to rectify the situation. In the event of incorrect reading from tape, for example, the tape is automatically positioned backward and reread. Persistent failure results in operator notification. The operator must decide then

whether to bypass the faulty record or to dump the contents of memory on the drum or on a special tape.

When an output operation is unsuccessful, the Master Control Program prevents destruction of the output information.

If an end-of-tape condition is indicated by the external-result descriptor, the Executive Routine locates the next input file or output tape and provides the required tape-handling procedures. In the case of an end-of-file condition, the Executive Routine refers to the object program for specific instructions.

**Summary.** Execution of a program-release operator by the object program causes the status bit of an input/output descriptor to be altered. This indicates to the Executive Routine that the referenced area is ready for processing. Provided an Input/Output Channel is free, an interrupt condition occurs and the input/output operation is initiated. The presence of a continuity bit in the descriptor enables the Executive Routine to cycle multiple read or write areas automatically. Completion of an input/output operation also causes an interrupt condition, to which the Executive Routine responds according to information provided by the external-result descriptor.

#### **HANDLING OF INTERRUPT CONDITIONS**

When any interrupt condition occurs, control is transferred automatically to the Executive Routine, which ascertains the cause and initiates program action. During compilation certain operators which cause interrupts are inserted in the object program to provide for necessary operations and to facilitate Multi-Processing. Their use in connection with input/output operations has just been described. They may also indicate a need for additional program segments or memory space, or the completion of a program.

Other interrupts are associated with standard program checks that determine such arithmetic conditions as overflow and underflow. These checks ensure that data is properly handled and that arithmetic operations do not violate the limitations of the hardware. Although these conditions occur infrequently, the operating system must be provided with a means of handling them.

Finally, there are error-condition and hardware-checking interrupts. These cause the operator to be notified of an equipment failure so that he may restart processing or take appropriate action.

Interrupt conditions may be classified as either Processor-dependent or Processor-independent. The

response of the Executive Routine, particularly in regard to the order in which it handles interrupts, depends upon which type has occurred. The interrupt conditions are listed below in these two categories; input/output interrupts, which have just been described, are not repeated here.

#### **Processor-Dependent Interrupts**

**PROGRAM COMMUNICATION.** This interrupt condition may indicate one of a variety of conditions: end of a program segment, end of a program, or the need for additional memory to accommodate data. The information required by the MCP to handle the situation is found in the top of the Stack.

**FLAG BIT ON.** If the program references a word containing a flag bit, an interrupt occurs. The MCP transfers control to an evaluation routine to determine the subsequent path of the program.

**PRESENCE BIT ON.** When a program or data descriptor is referenced, the status of the presence bit is automatically examined. If the bit is on it indicates that the information referenced is not in memory. The Executive Routine constructs an I/O descriptor, and as soon as an Input/Output Channel is available, transfers the information to memory. The presence bit in the descriptor is turned off and the program can be continued.

**DESCRIPTOR LIMIT.** If a program attempts to reference a word not specified by the descriptor, an interrupt occurs. This is a programming (specifically an indexing) error. The program is halted, the operator notified of the error, and the MCP transfers control to the Program Reject Routine.

**STACK OVERFLOW.** If the capacity of a program's Stack is exceeded, an interrupt occurs, permitting the MCP to adjust the size of the Stack. Control is returned to the program segment.

**DIVIDE CHECK.** If a divisor is found to be zero during a divide operation, the program is halted and the operator is notified. The MCP then transfers control to the Program Reject Routine unless otherwise advised by program option.

**INTEGER OVERFLOW.** When a store integer is executed and the pseudo-exponent is found to be a positive nonzero quantity, the program is halted and the operator is notified. The MCP converts the integer to a floating point number and returns control to the program. Program option can also be specified, such as ignoring overflow.

**EXPONENT OVERFLOW.** If an arithmetic operation results in an exponent greater than +63, the program is interrupted and the operator is notified and the MCP transfers control to the Program Re-

ject Routine. Program option can also be provided to allow the object program to continue processing.

**EXPONENT UNDERFLOW.** Any arithmetic operation resulting in an exponent of less than -63 causes an interrupt condition. The program is halted, the operator notified and the MCP then sets the result to zero and returns control to the object program unless otherwise advised by program option.

**INVALID ADDRESS.** If reference is made to a nonexistent address, an interrupt occurs. The Executive Routine examines the memory allocation tables (refer to Environment Control Routine) and makes appropriate adjustments to ensure valid memory references.

**MEMORY PARITY.** If a parity failure is detected during processing, an interrupt occurs. The Master Control Program notifies the operator so that corrective action may be instituted.

### **Processor-Independent Interrupts**

**TIME INTERVAL.** An automatic interrupt is initiated periodically, and the Executive Routine adjusts the operations log to reflect the elapsed time.

**PROCESSOR BUSY.** When the Processor is busy or inoperative, this interrupt occurs. If the Processor has failed, the operator is notified. If the Processor is busy, the Executive Routine refers to the operations log to determine what program is being run and whether it has exceeded its maximum processing time. If such is the case, the program is interrupted and the operator is notified.

**KEYBOARD.** The operator indicates that he wishes to transmit a message to the system by depressing the Inquiry key. As soon as an Input/Output Channel is available, the console signals that the system is ready to receive the message. The Executive Routine provides room in memory for storage of the message, and immediately resumes processing while the message is keyed in. The operator presses the End-of-Message key as soon as he is finished.

An extensive list of valid Keyboard messages is available, but operator communication is restricted to certain specified messages. This prevents indiscriminate use of the Keyboard and assures that only necessary information is requested. An example of a valid message is a request to enter a new job or to revise the schedule.

The Executive Routine determines the validity of a message. An unacceptable message is rejected automatically and the operator is advised. A valid message is analyzed and the appropriate program action is taken.

The Executive Routine transmits messages to the

operator via the Message Printer in response to Keyboard inquiries, to advise the operator of initial set-up requirements and processing needs, or to indicate component failure. Answers to valid Keyboard messages, for example, might include the name and running time of programs currently in memory. When a job is finished, the Executive Routine may request that input and output files pertaining to that program be removed, and that files for the next job be loaded. When the operator has fulfilled such a request, he transmits a compliance message via the Keyboard. The Executive Routine may also notify the operator of the location and cause of component failure, for example, a not-ready condition in any input-output unit.

Normal processing of program segments continues during communications. Once the Executive Routine has provided an Input/Output Channel and initiated the message printout, it returns control to a program segment.

**Program Reject Routine.** The Master Control Program provides a program reject routine to meet error conditions. The program in error may be bypassed, dumped on a special error tape or on the drum. If no dump tape has been provided, or if the drum contains program segments to be processed, the operator is requested, via the Message Printer, to indicate the program's final disposition. The Executive Routine bypasses the program until the operator provides a dump tape or decides to ignore the error condition and continue processing.

When a program is dumped, all information necessary to restart the program at the point of interruption is provided. A printout of the program contents may be made also. The printout includes the name and current value of each variable in the source language. There is also a record of the last statement label encountered in the program, and a listing of each statement label with the number of times it was encountered. This provides the programmer with a detailed, *source language* account of the status of his program. The operator is advised of program dumping via the Message Printer.

The programmer may write special error-detection routines to augment those provided automatically in the B 5000 System. These are referenced by the Executive Routine in lieu of its standard error handling procedures.

### **CONTROL OF PROGRAM SEGMENTS**

When processing of a segment is interrupted, the B 5000 automatically stores the address of the next syllable to be executed. After the interrupt condition has been satisfied, the address replaces the one in the



C register, effecting transfer of control to the proper point in the program.

If completion of a program segment causes an interrupt condition, the Executive Routine determines which segment is to be processed next. Control is transferred to a segment in memory of the highest priority job.

#### **USE OF OTHER MASTER CONTROL PROGRAM ROUTINES**

The Executive Routine loads and transfers control to the Schedule, Environment Control, and Error Routines when the need arises. If, for example, the operator requests a revision of job priorities, the Executive Routine responds by loading the Schedule Routine. Completion of any program signals the Executive Routine to load the Environment Control Routine so that it may allocate memory space and create control tables for the next job to be processed. When error conditions arise, the Executive Routine determines what type of error has occurred and selects and loads from the drum the special routine to handle the situation.

#### **MAINTENANCE OF AN OPERATIONS LOG**

A record is maintained of processing for each program: the maximum time required, the time the job was started, and elapsed running time. At any time during processing, the operator may query the Master Control Program to obtain the present elapsed running time for a job. When the total processing time is known, the operator may determine how much time is needed to complete a job. Such information is valuable if a new job is to be introduced to the system or if priority changes require a reordering of the current processing sequence. The log for each job is automatically recorded by the MCP and may be output optionally via the Message Printer or the Line Printer.

#### **MAINTENANCE OF SYSTEM DESCRIPTION**

The Executive Routine maintains a table of the physical system configuration at all times. The Schedule Routine references these parameters in determining which jobs may be combined in memory at once, and which processing sequence is most efficient. The Environment Control Routine uses the information when allocating memory to a given set of jobs. The table also permits the Executive Routine to adjust processing sequence when unforeseen circumstances arise. If, for example, a tape-unit failure occurs, the Executive Routine bypasses the program using the unit and advises the operator of the failure so that the file can be moved. When the file is re-located, the Executive Routine reassigns the unit

number in the Program Reference Table associated with the job, and returns to the normal sequence.

#### **SUMMARY**

The Executive Routine coordinates the functions of the Master Control Program, responds to all interrupt conditions according to their cause, maintains control over program segments and the other routines of the MCP, maintains a constant system description and operations log, and provides for system-operator communication.

#### **THE SCHEDULE ROUTINE**

This routine relieves the programmer and operator of all scheduling and ensures the most effective use of the system for multi- and parallel-processing. The operator loads program parameter cards, then makes a Keyboard request to schedule. The Executive Routine loads the Schedule Routine from drum to memory. Its functions are as follows:

1. To determine the sequence of jobs to be run and, if Multi-Processing is performed, the best combination of programs to be processed concurrently. Priority ratings, system requirements of each object program, and the present system configuration are considered.
2. To reschedule whenever a higher priority job is introduced. (Adjusting job sequence during production run is called dynamic rescheduling.)
3. To relay information about the jobs scheduled to the Environment Control Routine.

#### **PROGRAM BACKLOG TABLE**

The Schedule Routine develops a program backlog table so that it can perform these functions. The elements of this table are formulated from priority ratings provided by the operator and the program parameters produced by the compiler. Information furnished by program parameters includes job identification, input/output unit requirements, amount of memory required, file descriptions including identification and size, and object program media (cards or tape).

When all parts of the program backlog table have been developed it contains, in order of priority, each job identification, input/output requirements, memory requirements, current location of program (drum, tape or cards), file descriptors, processing status (running or finished), time started, time completed and input/output unit time. This table is maintained in memory throughout processing by the Executive Routine and is used by the Environment Control Routine for memory allocation and input/

output unit assignments.

### INPUT/OUTPUT REQUIREMENTS

The input/output requirements of each job being scheduled must be considered in relation to the present system configuration and to those of jobs scheduled for concurrent processing. Each Memory Module allows the following simultaneous input/output operations:

- One magnetic drum operation, or
- Two magnetic-tape operations plus any two of the following: Card Reader, Card Punch, Line Printer, Plotter, Keyboard, Message Printer.

For the purpose of analysis, the input/output devices are weighted according to their transfer rates. The Storage Drum is valued at 10, a Magnetic Tape Unit at 4, and all other devices at 1. Simultaneous access to one Memory Module is limited to operations with a total score of 10.

The number of Input/Output Channels available also determines the total number of simultaneous operations. For instance, one Magnetic Tape Unit, a Card Reader, Punch and the Keyboard have a total score of only 7, but will occupy four channels, assuming that a maximum system is available. Table 4-1 shows possible input/output combinations.

**Table 4-1. Input/Output Combinations With Maximum System**

DRUM	TAPE UNITS	OTHER DEVICES
1	0	0
0	2	2
0	1	3
0	0	4

Memory allocation (by the Environment Control Routine) is also influenced by the number of input/output units that must access a Memory Module. Whenever a system has more than one Memory Module, input/output areas are distributed in memory so that simultaneous operations affecting a particular Memory Module are kept within the restrictions.

### MEMORY REQUIREMENTS

When a program is compiled it is given a memory score that indicates the maximum number of locations it will require at any given time. The Schedule Routine considers this score in relation to the memory requirements of other jobs being scheduled and to the total capacity of the system.

### SUMMARY

In constructing the program backlog table the Schedule Routine first satisfies the priorities for the day's production. Then the input/output and memory requirements are examined. Jobs are grouped so that several can be processed at the same time; computing for one job is performed while input/output operations for the others take place.

As each job is completed, the processing status in the table indicates this. Finished items in the table contain all information for a comprehensive operations log.

### ENVIRONMENT CONTROL ROUTINE

By evaluating the information provided in the program backlog table in conjunction with the system description maintained by the Executive Routine, the Environment Control Routine makes assignments of input/output units and memory space.

### ASSIGNMENT OF INPUT/OUTPUT UNITS

Messages are transmitted to the operator, specifying the number and type of input/output files required for each program. Actual unit designation is not made yet. As soon as the proper files have been mounted, the operator indicates via the Keyboard that they are ready.

The Environment Control Routine scans each external device, locating the correct files, assigning unit numbers, and developing a record of these assignments. The input/output unit assignment table contains the number of every unit in the system, the file description (identification and size), title of the program using the unit, and the unit status. The status field indicates whether a job is in progress or finished or that the unit is free. A sample is shown in Table 4-2.

**Table 4-2. Input/Output Unit Assignment**

UNIT NO.	FILE DESCRIPTION	PROGRAM ASSIGNMENT	STATUS
1	File Title	Job Title	In Progress
2	File Title	Job Title	In Progress
3	Blank	Job Title	In Progress
4	Blank	Blank	Unit Free
5	File Title	Job Title	Finished

### ALLOCATION OF MEMORY

Once the input/output assignment table has been generated, the Environment Control Routine allo-

ates memory space for the Program Reference Table, Stack, segments, data arrays and input/output areas of each program. It refers to the memory scores and file descriptions so that areas of the right size can be reserved. Input/output requirements are referenced so that assignment of input and output areas concurs with the restrictions on simultaneous operations affecting each Memory Module.

For the purpose of allocation, memory is divided into many blocks. The Environment Control Routine develops a memory allocation table (Table 4-3) that indicates which blocks are used by each object program and which are unassigned. A record is made of blocks that are set aside for program use but not yet filled, and for those that are completely filled with information. This record is maintained and referenced throughout processing.

Now the Program Reference Tables are loaded and initialized. Core base addresses are supplied to all the descriptors on the basis of assignments that have just been made.

**Table 4-3. Memory Allocation**

MEMORY BLOCK	CONTENTS
1	Job 1, Program Reference Table
2	Job 1, Program Segments
3	Job 1, Input/Output Area
4	Job 1, Still Unused
5	Job 2, Program Reference Table
6	Job 2, Still Unused
7	Job 2, Program Segments
8	Job 3, Program Reference Table
9	Job 3, Program Segments

**BASE LOCATION TABLES**

The Environment Control Routine produces a table that lists the base location of the Program Reference Table for each job (Table 4-4) and one that lists the base location (within the Program Reference Table) of the input/output descriptors for each job (Table 4-5). These tables are used by the Executive Routine to control Multi-Processing. Specific uses are described in connection with input/output operations.

**LOADING PROGRAM SEGMENTS**

The Environment Control Routine loads the program segments required to start processing. The presence bit is set in the corresponding program descriptors in the Program Reference Table. The number of segments located in memory for each job at any one time depends upon the nature of the

program, size of data arrays and input/output areas, and available space. Now the Executive Routine determines the processing sequence from information in the program backlog table and transfers control to a segment.

**Table 4-4. Base Locations of Program Reference Tables**

JOB	BASE ADDRESS OF PROGRAM REFERENCE TABLE
Job 1	XXXX
Job 2	XXXX
Job 3	XXXX
Job 4	XXXX

**Table 4-5. Base Locations of Input/Output Descriptors**

JOB	NUMBER OF INPUT/OUTPUT DESCRIPTORS	BASE LOCATION (WITHIN PROGRAM REFERENCE TABLE)
Job 1	2	XXXX
Job 2	3	XXXX
Job 3	2	XXXX
Job 4	1	XXXX

**LOADING ADDITIONAL SEGMENTS**

The sequence just described is the job set-up procedure. As programs are processed or finished the Executive and Environment Control Routines perform other operations. When, for example, a program descriptor is referenced and the presence bit indicates that the segment is not in memory, the Executive Routine refers to the memory allocation table. Certain blocks may have been set aside for a job, but are as yet unused. In this case, the Executive Routine selects an empty block, supplies the base address to the program descriptor, loads the segment into the area, and returns control to the object program. If, however, all the blocks allocated for a job have been used, the Environment Control Routine is loaded. Provided there is available space, it assigns a new memory block to the job and loads the segment; otherwise, it provides an overlay area in a block previously used.

**PROGRAM-FINISH CONDITIONS**

When a job is completed, the Environment Control Routine scans the control tables, adjusting all entries pertaining to that job. The next program to be run

is located in the program backlog table. The Environment Control Routine makes unit and memory assignments as described; advises the operator of file requirements; loads and initializes the Program Reference Table and the base location tables; loads the segments and transfers control to the new program.

#### **CHANGING THE SCHEDULE**

If a new job is introduced to the system with a high

priority rating, the Schedule Routine is loaded and alters the program backlog table. The Environment Control Routine makes the proper unit and memory allocations. If sufficient memory space is not available, a lower priority job is removed from the system after restart points have been stored for later processing. The normal procedure for initializing and loading a program is then followed.



# SECTION 5

## COMPONENTS

### INTRODUCTION

The system chart in Figure 5-1 illustrates the remarkable flexibility of the B 5000 Information Processing System. This flexibility is called Program Independent Modularity because the configuration of the system can be altered without causing disruption in processing or requiring program modification. Three types of modules make up a system: Processors, Input/Output Channels, and Memory Modules. These may be coordinated into a system tailor-made for a user's requirements. A system may incorporate one or two Processors, as many as four Input/Output Channels, and as many as eight Memory Modules. Each Input/Output Channel can communicate with any Memory Module, and each Processor can also communicate with any Memory Module. This basic network of communication is augmented by the ability of each Input/Output Channel to control information flow for any of the 26 input-output units, such as Magnetic Tape Units, Storage Drums, Card Readers, Line Printers, etc., that can be included in a maximum system.

The maximum and minimum system configurations are itemized in Table 5-1.

Table 5-1. System Configuration Limits

COMPONENT	MINIMUM	MAXIMUM
Console	1	1
Processor	1	2
Memory Module	1	8
Input/Output Channel	1	4
Storage Drum	1	2
Magnetic Tape Unit	0	16
Line Printer	0	2
Card Reader	1	2
Card Punch	1	1
Plotter	0	1
Keyboard	1	1
Message Printer	1	1
Input/Output Exchange	1	1
Memory Exchange	1	1

Each component is described in the following paragraphs. The Processor, which has been discussed extensively in Section 3, System Characteristics, is not described further here.

### CONSOLE

The Console, shown in Figure 5-2, is the operations center of the B 5000 system. It has control switches and indicator lights which provide the operator with a convenient supervisory center. Next to and considered part of the Console control center are the Message Printer and Keyboard, the communication links between the system and operator.

The Console also contains the system switching facilities, the interrupt circuitry, the system synchronization equipment, the power controls, and the interval timer or clock. The interval timer provides a constant interval time interrupt for control purposes. This timer allows the logging of running time of all programs.

### MEMORY MODULE

The high-speed Memory Module is the primary storage unit for programs and data. It also couples the computation facility of the Processors with the input/output operations. It is able to accept and transmit data independently of the Processor's activity. In addition, the Processor can access a Memory Module, even though the module is currently communicating with input/output devices.

Each module can store 4096 words. A single system may have from one to eight modules, each independent of the others. This independence facilitates multiple and parallel processing. When a system has one Memory Module, for example, multiple Processor and input/output operations are time-shared in memory. Processor operations can resume as soon as the input/output operation has been *initiated*, so there is virtually no delay. If two or more Memory Modules are available to the system, parallel input/output and Processor operations can be accomplished with no delay.

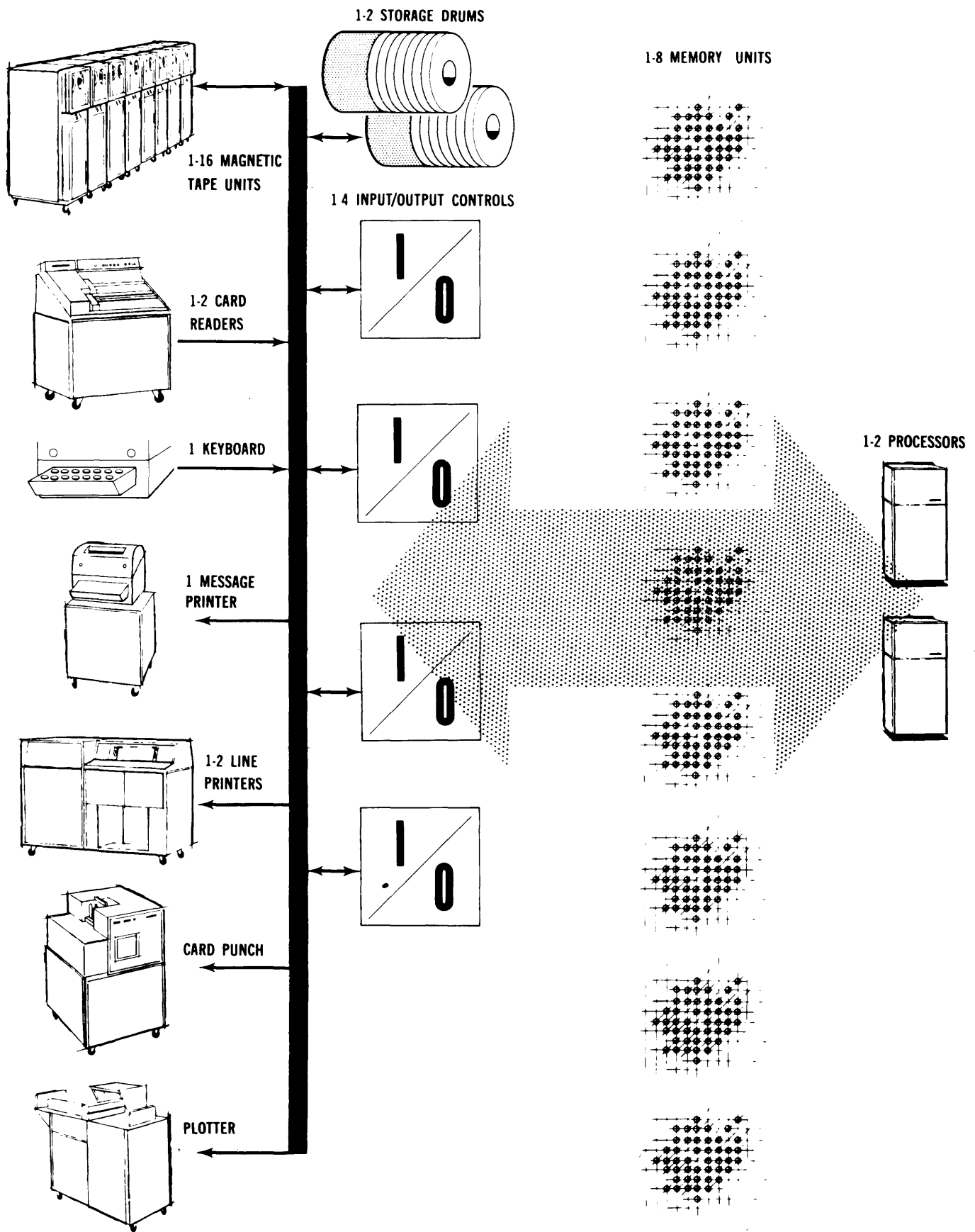


Figure 5-1. System Configuration Chart



Figure 5-2. The Console

The Memory Module, shown in Figure 5-3, is a coincident current, magnetic-core type, with a random access time of 3 microseconds ( $\mu\text{s}$ ) per word and a memory cycle time of 6  $\mu\text{s}$ . Its memory-retention feature allows information to be stored indefinitely without regeneration. Each of the 4096 words contains 48 bits of information and 1 parity bit. A 49-bit Memory register permits information to be transferred in parallel between either the Processor or the Input/Output Channels and the Memory Modules. A storage location within a module is addressed by a 12-bit Memory Address register associated with each Memory Module. One word may be addressed during each memory cycle; any address or combination of addresses may be used continually.

### MEMORY EXCHANGE

This is a switching network which provides automatic hookup of either Processor to any Memory Module and any Memory Module to any Input/Output Channel.

### INPUT/OUTPUT EXCHANGE

This is a switching network, similar to the Memory Exchange, that allows any one of the Input/Output Channels to communicate with any one of the input/output units.

### INPUT/OUTPUT SYSTEM

Information may be transmitted between Memory

Modules and the Card Punch, Card Readers, Line Printers, Magnetic Tape Units, Storage Drums, the Keyboard, the Plotter, and the Message Printer.

A maximum of four Input/Output Channels may be incorporated in a system; the maximum number of each type of input/output device is listed in Table 5-1.

Normally, all input and output operations transfer information directly to or from memory, utilizing a time-sharing technique. This technique allows computation and input/output operations to proceed simultaneously, except when both the Processor and any Input/Output Channel are communicating with the same Memory Module at the same instant. In this case, the Processor accesses memory between input/output-memory word transfers. The fast cycle time of six  $\mu\text{s}$  permits virtually simultaneous access; for instance, it is possible for the Processor to maintain maximum-speed computation while four input/output operations are conducted (through the four channels) at maximum speeds.

The program is interrupted briefly at the completion of each input/output operation. At this time, the Master Control Program initiates a new input/output operation and returns control to the program. Thus the B 5000 System controls all phases of operation with maximum efficiency. Because the external operations can be initiated, then carried out without halting the Processor, inefficiencies due to component idleness are eliminated.



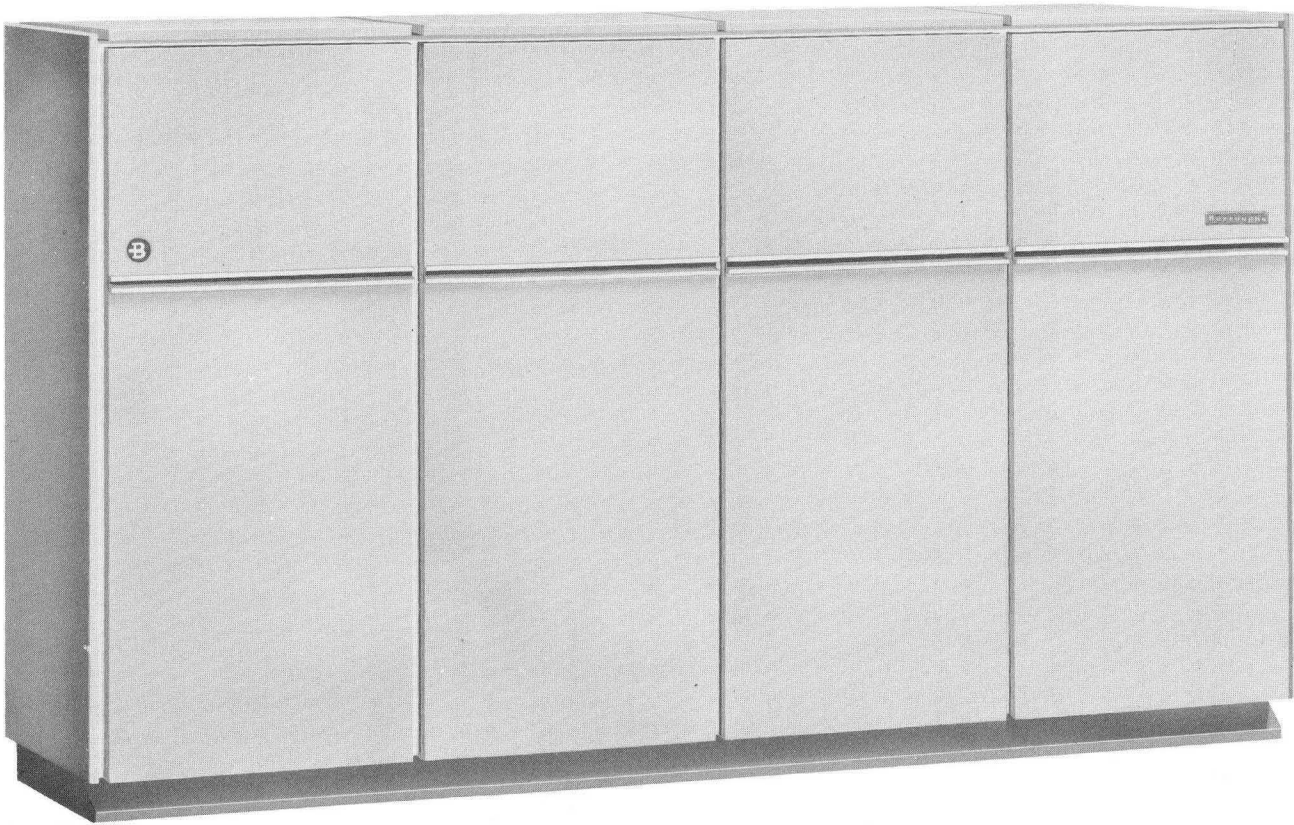


Figure 5-3. Memory Modules and Input/Output Channels

### INPUT/OUTPUT CHANNEL

An Input/Output Channel, shown in Figure 5-3, handles the flow of information in one direction at a time between any Memory Module and any peripheral device attached to the system. The number of channels included in a system determines the number of simultaneous input/output operations. If there are four Input/Output Channels, for example, there can be as many as four simultaneous input/output operations.

If one channel is sufficient to handle the volume of input/output operations, all devices available may be associated with this channel. If increased workload makes necessary a second Input/Output Channel, the same input/output devices are available to it. The increase in system capacity is immediately available; no reprogramming is required.

### INPUT INFORMATION FLOW

Input information flows through an Input/Output Channel by way of a 48-bit Buffer register. This register accepts one character at a time from the input device. The six bits of information that make up each character are transferred in parallel. When a complete word (48 bits) has accumulated in the Buffer register, it is transferred in parallel (simul-

taneously) to memory.

The information being transferred may be either alphanumeric or binary. Alphanumeric information is in the six-bit BURROUGHS Common Language (BCL) code, and is obtained from even-parity magnetic tape, from one card column that has been decoded by the Card Reader, or from the Keyboard, also decoded. Binary information is obtained from odd-parity magnetic tape, storage drums, or a column of binary code on punched cards. The differences in magnetic-tape parity for BCL and binary code are explained in this section under the heading Magnetic Tape. Figure 5-4 is a flow diagram of the input information flow.

### OUTPUT INFORMATION FLOW

Output information flow is the reverse of input. Each word is transferred in parallel (all 48 bits simultaneously) from a Memory Module to the Buffer register. It is sent six bits in parallel, a character at a time, to the output device. Binary output information is transferred either to a Storage Drum or to a tape as odd-parity data. Alphanumeric information may be transferred to even-parity tapes, Line Printers, the Card Punch, or the Message Printer. Figure 5-5 illustrates this flow.

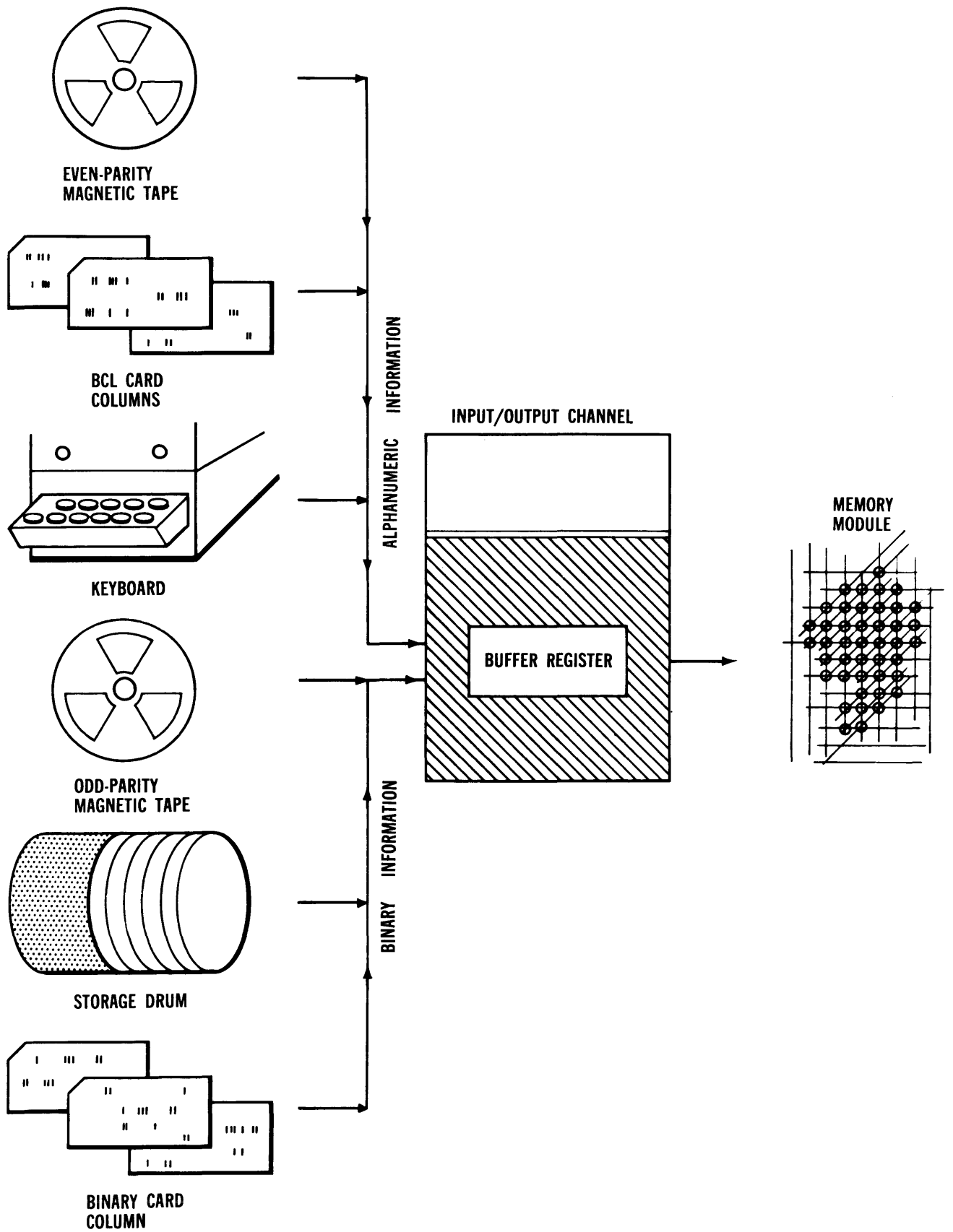


Figure 5-4. Input Information Flow

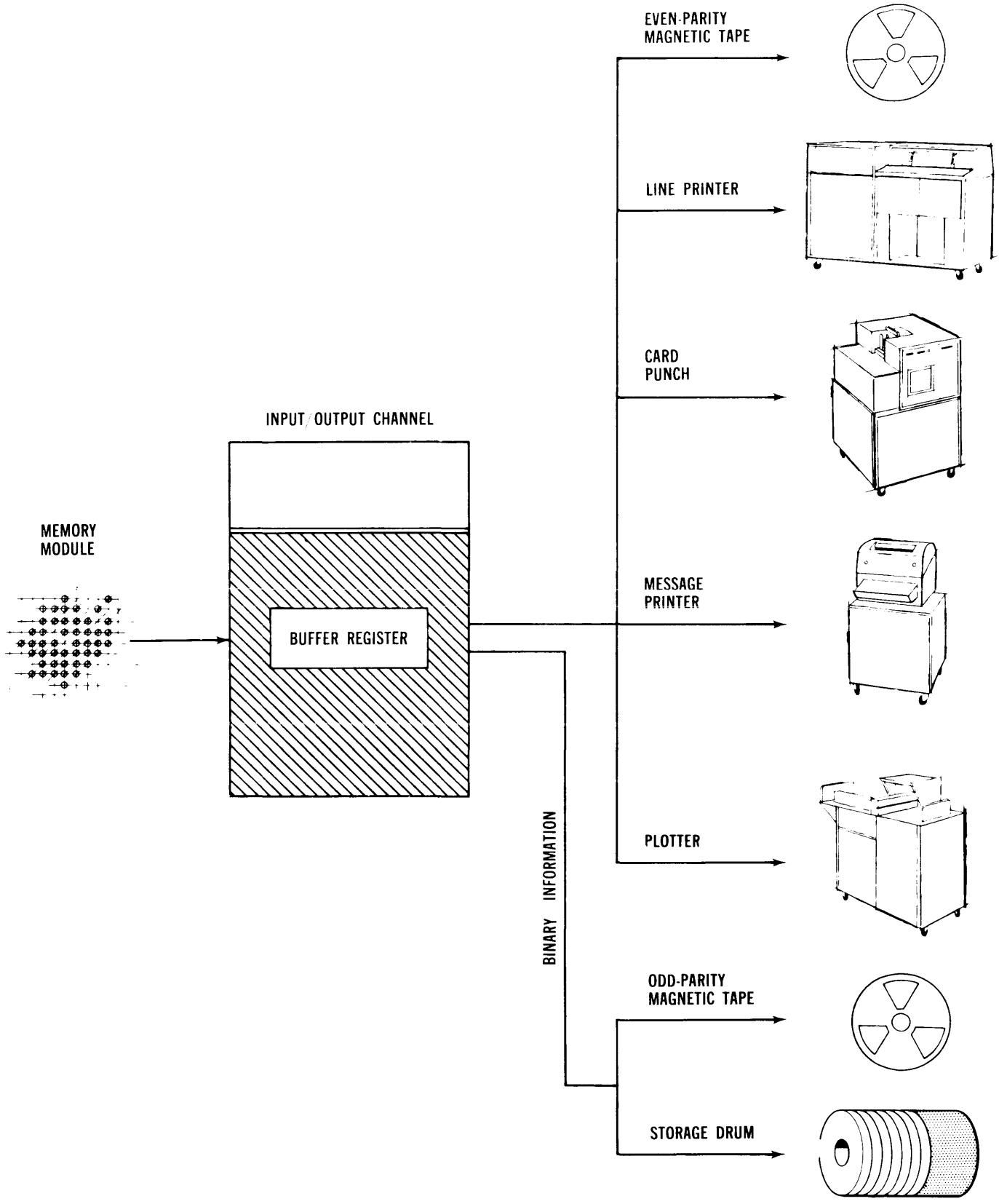


Figure 5-5. Output Information Flow

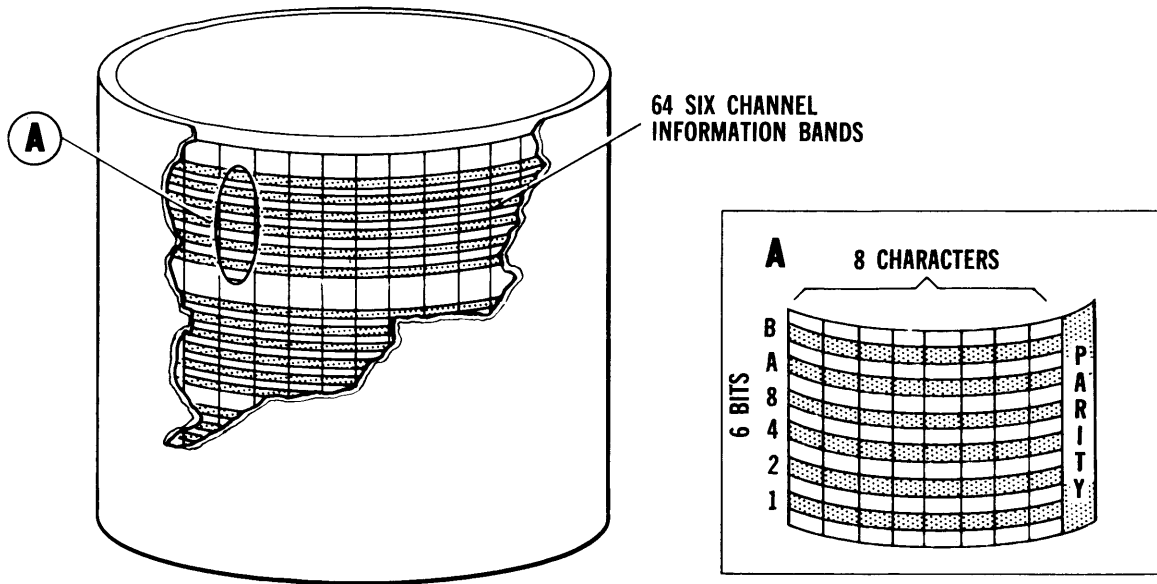


Figure 5-6. Storage Drum

## STORAGE DRUM

The Storage Drum is a high-speed mass storage device providing rapid access to program segments, subroutines, and large blocks of data. It communicates with Memory Modules through an Input/Output Channel. Transfers between the Storage Drum and Memory Modules are made independently of Processor activity, permitting simultaneous computing and input/output operations. The basic B 5000 System includes one Storage Drum; a second drum may be added.

Figure 5-6 shows the 64 six-channel information bands, each containing 512 words. Total storage capacity is 32,768 words. The drum rotates at a rate of 3600 rpm, and has a read-write speed of 8.1 microseconds per character. Average access time to a word on the drum is 8.5 milliseconds or 18.5 milliseconds if switching bands. Information can be retained on the drum indefinitely without regeneration.

As shown in Figure 5-6, a longitudinal parity of six bits is recorded at the end of each 48-bit word. The transfer of information between Storage Drum and an Input/Output Channel is made character by character, while transfers between the channel and Memory Modules are by the word. Each word on the drum is addressable. Reading or writing may start at any drum address. From one to 512 words may be transferred.

Automatic lane advancing is not possible. An attempt to record data in the location immediately after the last (512th) location of a lane causes the word to be recorded in the first location of the same lane. Thus, a 20-word record begun in the 500th location is carried over through the first eight locations of the same lane. The 512 words of a lane are

written as two interlaced groups of 256 words. Thus any word is addressed in one revolution of the drum; a continuous reading or writing of 512 words requires two revolutions.

Four manual switches lock out the lower half of the drum addresses. Each switch protects 4096 words to prevent inadvertent overwriting of the Master Control Program, Language Translators and Compilers, and Utility Routines which are permanently recorded in this area.

## MAGNETIC TAPE UNIT

The B 5000 magnetic tape system operates independently of the Processor. Reading, writing, backspacing, rewinding, and erasing operations are under system control. A maximum of 16 of the Magnetic Tape Units shown in Figure 5-7 may be used with the system.

The B 5000 Magnetic Tape Unit accepts data in either binary or single-frame alphanumeric form. Tape format is compatible with IBM Model 729-II and 729-IV magnetic tape units. Standard tape one-half inch in width is used. Tapes are mounted on reels which contain up to 3600 feet of tape and have a maximum diameter of 10½ inches.

Data may be stored in two densities, either 200 or 555.5 frames per inch. One frame contains either six binary bits or one six-bit alphanumeric character. Tape speed is 120 inches per second, a transfer rate of 24,000 characters per second for a density of 200 frames per inch, and 66,600 characters per second for a density of 555.5 frames per inch. Packing density is selected by a switch on the unit.

Tape is rewound at a speed of 340 inches per second. Start or stop time is 5 ms. The dual-gap read-write

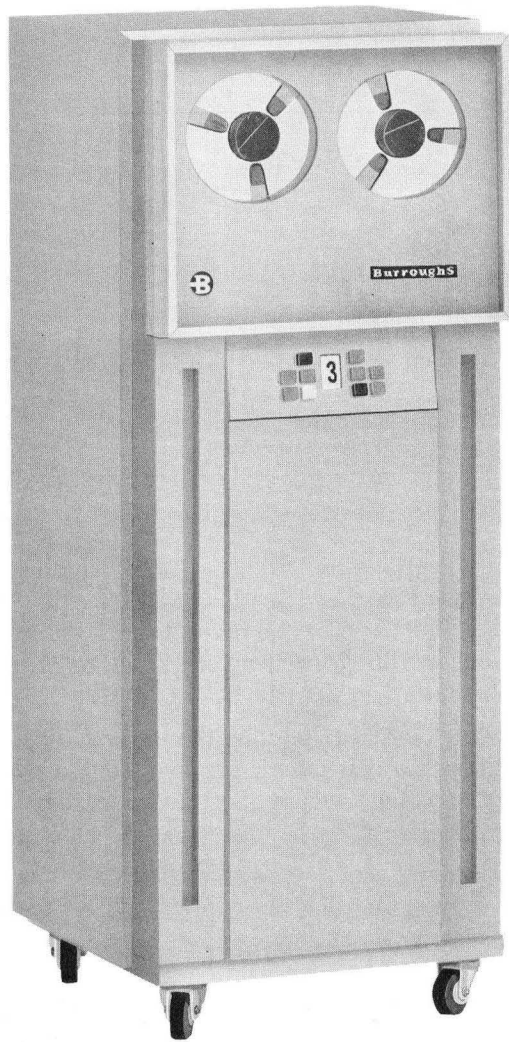


Figure 5-7. Magnetic Tape Unit

head of the B 5000 Magnetic Tape Unit provides automatic checking of write operations.

The Magnetic Tape Unit operates in two modes: local or remote. Local manual control of the unit is provided by a Local-Remote switch on the operator's panel. An indicator denotes which operating mode is in effect.

Mounting and removal of tape reels is facilitated by quick-action reel locks. A Reel Brake Release switch permits loading or unloading of the tape. A write ring must be installed on a reel to permit writing or erasing, thus preventing accidental destruction of files. After the tape reel has been mounted on the unit, activating the Load switch causes the tape to be drawn into the slack loop mechanism, and to be automatically positioned at the beginning-of-tape marker, ready for operation. An Unload switch is used to reverse the load procedure for removal of the

tape reel. The unit must be in local mode for these operations.

A Ready indicator shows when the transport is in a ready state. A Write Warning light is turned on if the reel installed on the transport is equipped with a write ring.

Tape format consists of seven recorded channels across the tape, as illustrated in Figure 5-8. The information tracks, identified as 1, 2, 4, 8, A, and B, represent either single-frame alphanumeric or binary information. The C track provides a parity check for each frame. The non-return-to-zero method is used for recording.

Reading and writing can be done in either binary or alphanumeric mode, providing complete code flexibility to the system. The alphanumeric mode carries an even parity. That is, a parity bit is recorded in the C channel simultaneously with each character if an odd number of bits represents that character in the information channels. The binary mode carries an odd parity. A parity bit is recorded in the C channel if there is an even number of bits in the information frame.

It is possible to write interspersed binary and alphanumeric records in the binary mode.

A longitudinal check character is written after the last character of each record. This consists of a parity bit, automatically recorded in each track with an odd total bit count. These parity bits maintain an even number of one-bits in each track for the entire record length, regardless of the code used.

### READ OPERATIONS

Both forward and backward read operations can be performed in either binary or alphanumeric mode. Regardless of reading direction, information is placed in memory in normal sequence. Variable-length records ranging from one character to memory size can be read in one-character increments in the alphanumeric mode. In the binary mode, one to 1024 words may be loaded in one-word increments.

Reading operations in either mode are terminated when a  $\frac{3}{4}$ -inch record gap is encountered. A group mark is automatically inserted in memory after the last character of an alphanumeric record. In either mode, if the number of characters read (including the group mark in the alphanumeric mode) does not complete a word, the information appears in memory left-justified on forward read.

### WRITE OPERATIONS

Write operations are performed in the forward direction, placing information from memory in ascending sequence on tape. Alphanumeric records may vary

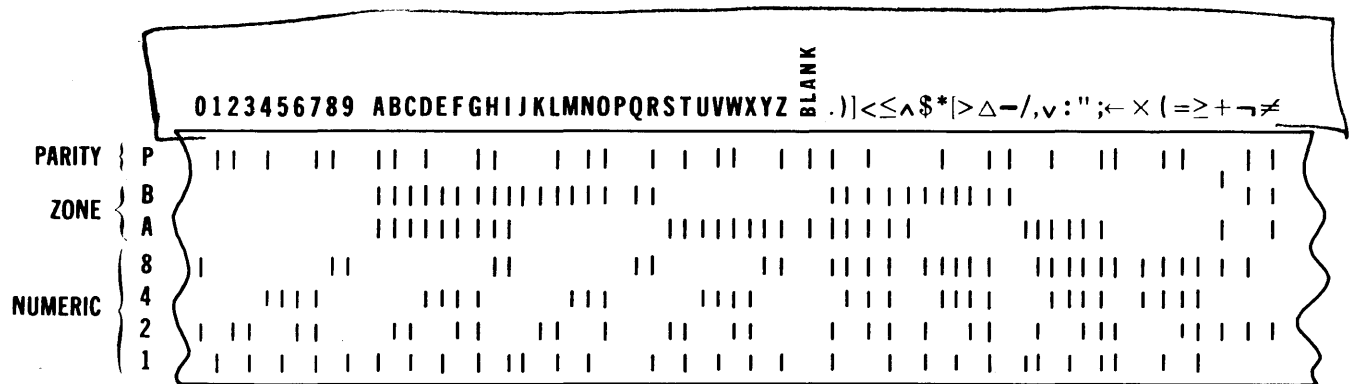


Figure 5-8. Magnetic Tape Format

in length from one character to memory size in one-character increments. A group mark terminates an alphanumeric write operation. When the binary mode is used, one to 1024 words may be recorded in one-word increments. Writing the number of words specified by an output descriptor terminates a binary write operation. A  $\frac{3}{4}$ -inch record gap is automatically supplied at the end of any write operation.

The address of the last word read into or written out of memory in both binary and alphanumeric modes is recorded in an external result descriptor. The descriptor is supplied to the system from the Input/Output Channel at the completion of a write operation, and indicates to the Master Control Program the status of that operation.

In addition to reading and writing, the tape system performs backspacing, rewinding, and erasing operations. When encountered, a backspace descriptor causes the designated tape unit to backspace one record. A rewind descriptor causes the unit to rewind to the beginning-of-tape marker. An erase descriptor is used to erase desired lengths of tape, as in extending a record gap over a flaw.

When an end-of-file occurs, a special one-character record is written. It consists of a bit in each of the 1, 2, 4, and 8 tracks, and is always followed by a longitudinal check character with the same bit structure. The A, B, and C tracks contain zero bits. This

mark is recorded in alphanumeric mode, regardless of which mode was used to write the information records. It follows the standard  $\frac{3}{4}$ -inch record gap after the last record in the file.

The beginning-of-tape marker is located approximately 10 feet from the physical beginning-of-tape. The end-of-tape marker is approximately 14 feet from the physical end-of-tape. Each is a one-inch strip of reflective tape, adhering to the plastic side of the magnetic tape.

Beginning- and end-of-tape markers are sensed photoelectrically. When the beginning-of-tape marker is sensed, the tape stops, whether it has been moving in a forward or backward direction. It can move only in a forward direction on the next operation. The end-of-tape marker can be sensed only when the tape is moving in the forward direction. Sufficient writing area is available, however, so that any writing operation in progress may be concluded.

## LINE PRINTER

The Line Printer shown in Figure 5-9 operates at 650 lines per minute. There are 120 print positions per line, and 63 characters plus a blank are available for each print position. Two optional character sets are available to provide complete flexibility for all printing requirements. There are 10 characters per inch horizontally, and either six or eight lines per

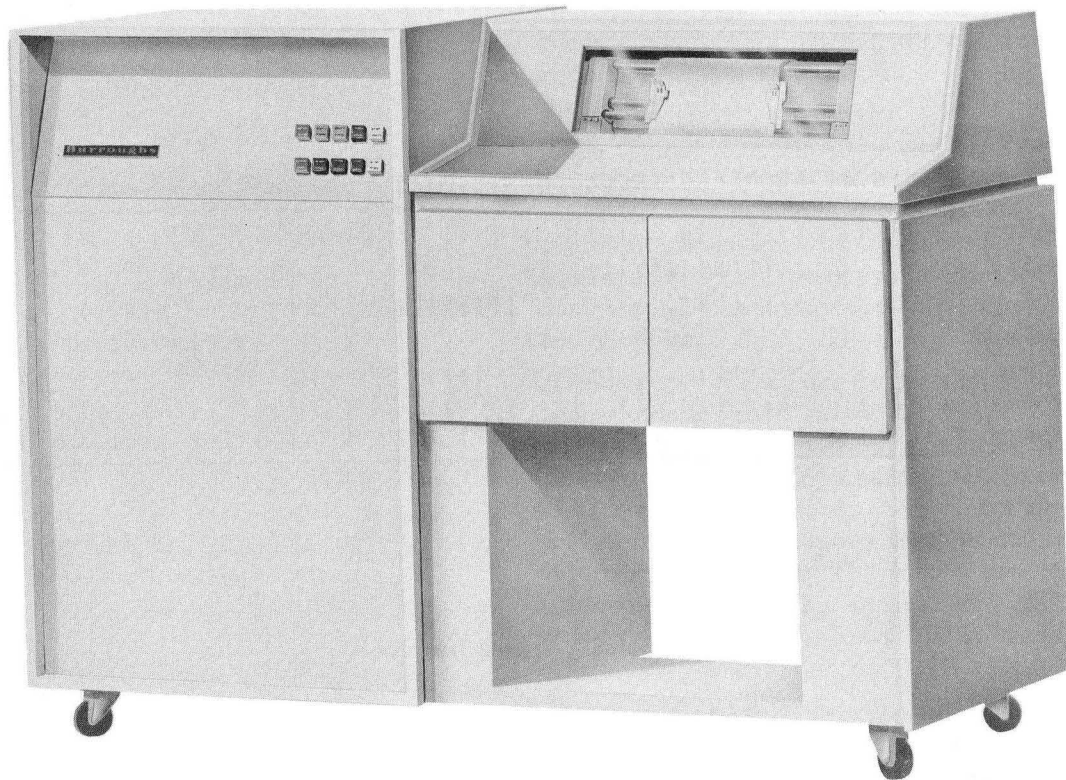


Figure 5-9. Line Printer

inch vertically. A maximum of two printers may be installed with one system.

The printer accepts 15 words of BURROUGHS Common Language alphanumeric information for each line of print. This information is transferred from ascending memory locations a character at a time through an Input Output Channel and accumulates in the 120-position buffer of the printer. When the buffer is completely loaded, the line is printed. Access to a print cycle is immediate.

Continuous paper forms are used. They may be from 5 to 20 inches wide, including margins, and a form may have a maximum length of 22 inches. Form adjustments require no special tools and drum clearance may be adjusted. Each tractor mechanism which controls horizontal placement of the form can be adjusted independently when the machine is stopped. Once installed, the form can be shifted left or right in minute increments while the machine is operating. Precise vertical form adjustments can be made in either direction when the printer is stopped. Processed forms are stacked.

Legible printing can be produced on forms from .0025

to .020 inches thick. Up to six-part forms on white sulfite bond can be printed, using .001-inch carbons. This number can be increased by using premium papers and carbons.

A vertical format punched tape, working in conjunction with the system, controls the vertical format of the printing, including such operations as skipping to a new page or skipping lines within a page.

The Line Printer produces an end-of-page signal, a print-check signal, and a not-ready signal.

Print checking consists of a parity check when characters are read into or out of the print buffer. Drum synchronization is checked by means of the drum position counter to assure that the drum position and timing circuits agree.

### CARD-HANDLING EQUIPMENT

A maximum of two Card Readers and one Card Punch may be attached to a system. They operate independently of the Processor and of any other peripheral equipment. As with all input/output devices, the card equipment can be selected by any

Input/Output Channel. Two Card Reader models are available. Both can handle either alphanumeric or binary information. (Binary information on cards is represented as 12 bits per column, and is read by column.) The Card Punch handles alphanumeric information.

### CARD READERS

Figure 5-10 illustrates the two Card Readers available. One model operates at 800 cards per minute, the other, known as the Program Card Reader, at 200 cards per minute. Each has an immediate-access clutch to eliminate clutch access time. Reading is performed photoelectrically by 12 photodiodes.

The 800 cpm reader can read cards of 51, 60, 66, or 80 columns. The Program Card Reader handles 80-column cards only. Both readers accept two thicknesses of cards. A card file, however, must be consistent in column length and card thickness. The information is read serially, column by column. Card data may be represented in standard tabulating card code or straight binary code. Each card file must contain cards prepared in only one code.

Standard card code is translated into BURROUGHS Common Language six-bit code, and supplied to the Input/Output Channel a character at a time. Invalid characters are sensed and replaced by six zero bits and an error indication is supplied to the Processor. Binary codes are read column by column, six bits at a time. Since there are 12 bits in a binary column, each card column occupies the equivalent of two six-bit characters.

The read circuitry is monitored during each card cycle, and when an error is detected an indicator is turned on. The operator may signal an end-of-file condition by depressing the Card Reader End-of-File key, when the card hopper is emptied. The reader may then run out the two or three cards still inside the reading mechanism and provide an end-of-file signal to the Input/Output Channel.

The hopper and stacker of the Program Card Reader each have a capacity of 500 cards. Those of the 800 cpm reader have a capacity of 2400 cards each. Both reader models allow removing or adding cards while the reader is operating.



Figure 5-10. 800 CPM Reader and Program Card Reader



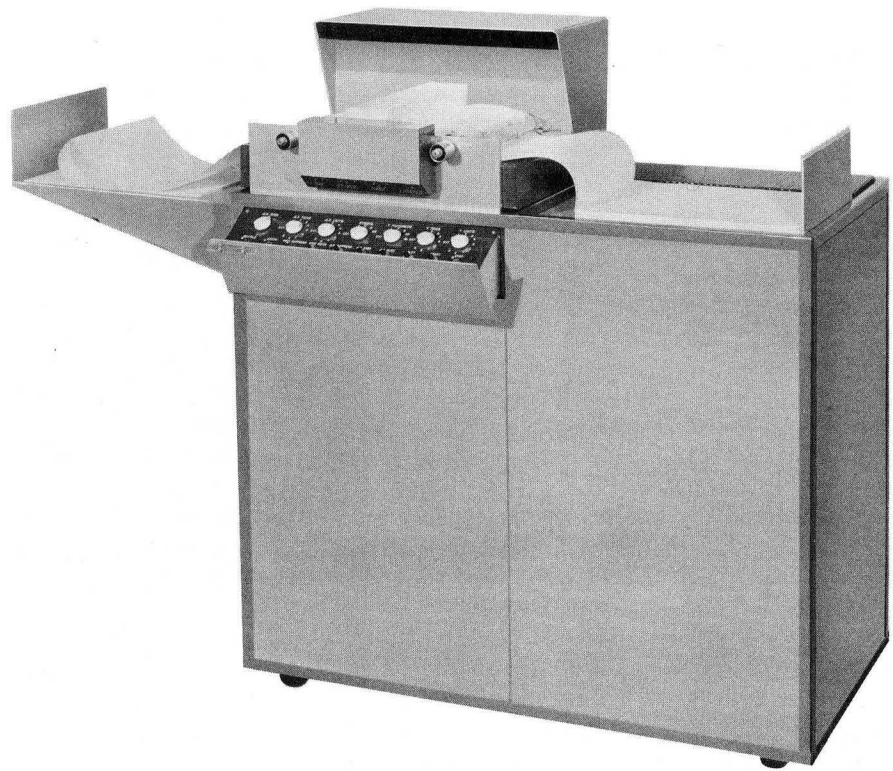


Figure 5-12. Plotter

### CARD PUNCH

The Card Punch (Figure 5-11) will feed, punch, check, and stack 80-column cards in both standard and postcard thicknesses at a maximum rate of 100 cards per minute. Functional controls are located on the plugboard and on the operator's control panel.

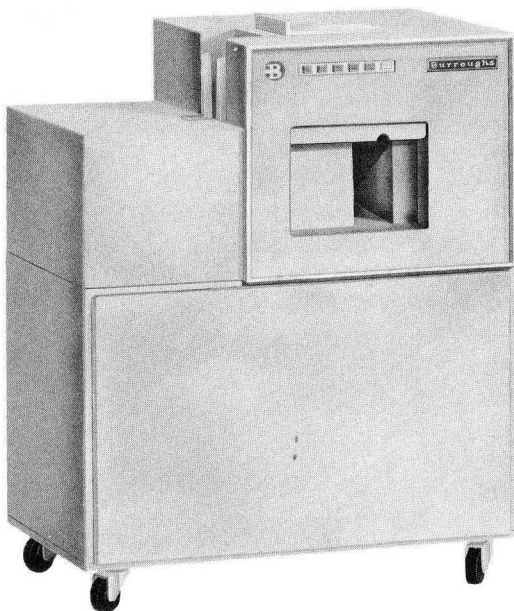


Figure 5-11. Card Punch

Double-punch and blank-column detection units are available in groups of 20 as optional devices. The punch contains a single-panel plugboard for the wiring of double-punch and blank-column checking. The hopper and stacker can hold 800 cards each.

### PLOTTER

The Model 201 Tally Digital Plotter (Figure 5-12) provides immediate visual display of computed results. It plots up to 8 points per second with selected symbols, or up to 20 points per second with random symbols. The vertical axis may be a maximum of 10 inches; the horizontal axis may be as long as desired.

Plotting resolution is 400 points in 10 inches, with the points spaced .025 inch apart on the vertical (Y) axis. The horizontal (X) axis has paper-feed spacing of  $\pm .025$  inch per increment of X, with zero to 99 increments available per input. Four symbols are used: period, square, triangle, and inverted triangle.

Grid printing is optional and may be suppressed when preprinted paper is used. A vertical grid strip  $\frac{1}{4}$  inch wide is printed whenever the grid system is activated; this occurs every time the paper advances ten increments along the horizontal axis. Any of the digits 0 through 9 may be printed slightly below the "O" abscissa axis. Manual controls over Plotter operation include stop, reset, paper step, and paper feed.

## MESSAGE PRINTER/KEYBOARD

The operator and the B 5000 system communicate via the Message Printer and associated Keyboard. The system thus has the means to instruct the operator and to provide answers to the operator's inquiries. The operator may acknowledge the instructions transmitted to him by the Master Control Program and initiate inquiries and instructions to the Master Control Program.

### MESSAGE PRINTER

The Message Printer (Figure 5-13) prints 600 characters per minute, a character at a time. The 63 characters plus blank of the BURROUGHS Common Language Code are used. The printer records all information initiated at the Keyboard, as well as that transmitted by the Master Control Program, thus providing a permanent record of operations.

### KEYBOARD

The Keyboard is also shown in Figure 5-13. The Keyboard is similar to a typewriter, containing 26 alphabetic characters, 10 numeric characters, and 3 special characters. The special characters are as follows:

CHARACTER	MEANING
Δ	Change Mode
-	Minus
(Blank)	Space

There are three function keys. The Inquire key interrupts the system so that a message can be transmitted to the Master Control Program. The End-of-Message key produces a six-bit group mark code. The Error key produces a six-bit invalid character code.

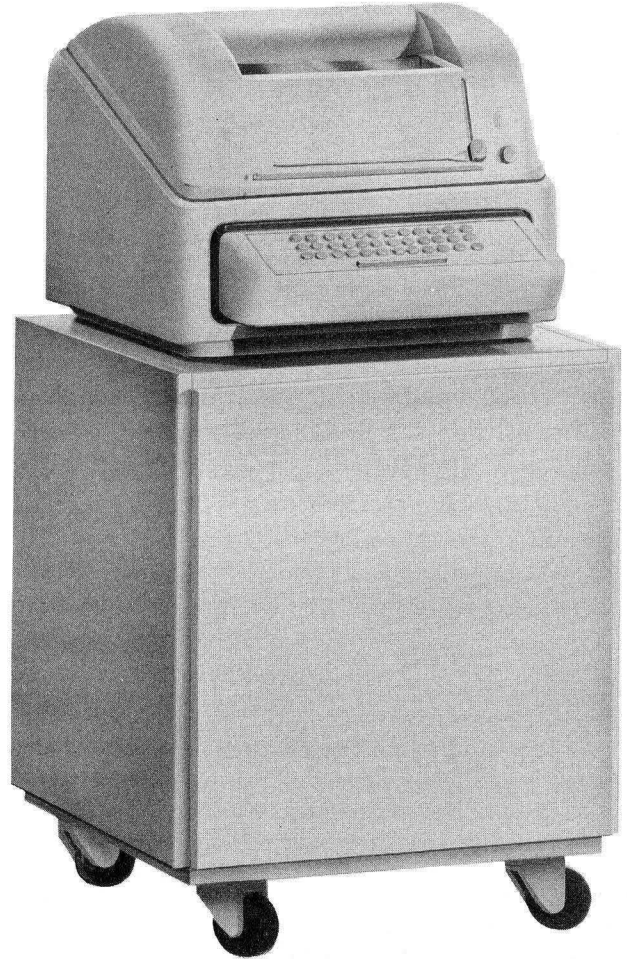


Figure 5-13. Message Printer/Keyboard



# APPENDIX A

## ALGOL CHARACTERISTICS

### INTRODUCTION

For problems essentially computational or mathematical in nature, the language of the B 5000 contains ALGOL 60 with minor restrictions. ALGOL 60, like any language, is formulated from a set of basic symbols and words. These symbols and words are then used to formulate expressions, which in turn are compounded into the programming units of the language called statements. The statements, supported by declarations, are computing instructions to the B 5000 and its compiler.

The purpose of this appendix is to define the terminology of ALGOL 60, showing its efficiency as a programming tool. It is not intended as a computational programming primer.

### BASIC SYMBOLS AND WORDS

The alphabet of ALGOL 60 comprises basic symbols (letters, digits, operators, etc.) and some English words. The words of ALGOL 60 are the identifiers, numbers, and strings which are formed from the alphabet.

#### LETTERS

The 26 capital letters of the English language are used to form identifiers (names or labels for variables, switches, etc.) and strings (special groups of basic symbols):

A B C D E F G H I J K L M N O P Q R S T U V  
W X Y Z

#### DIGITS

The ten Arabic digits (0-9) are used, both to form numbers and in arbitrary combinations with alphabetic characters to make up identifiers and strings.

### OPERATORS AND SYMBOLS

The following arithmetic operators are acceptable.

+ addition  
- subtraction  
× multiplication  
/ division  
**DIV** division (both operands of type **INTEGER**)  
\* exponentiation

The relational operators are:

< less than  
≤ less than or equal to  
= equal  
≥ greater than or equal to  
> greater than  
≠ not equal

The logical (Boolean) operators are:

**EQV** logical equivalence (equivalent)  
**IMP** logical implication (implies)  
∨ logical sum (or)  
^ logical product (and)  
¬ logical negation (not)

The sequential operators are: **GO TO**, **IF**, **THEN**, **ELSE**, **FOR**, and **DO**.

The separator symbols are:

, comma; separates arguments of a function, separates elements of a list, subscripts of an array, etc.  
. decimal point  
: colon; follows a label  
; semicolon; separates statements  
← replace by

plus **STEP**, **UNTIL**, **WHILE**, and **COMMENT**.

The bracket symbols are:

( ) Parentheses are used, for example, to enclose parameters of a procedure.  
[ ] Brackets are used, for example, to enclose subscripts of an array.  
" Quotation marks are used to enclose the symbols of a string.

The reserved words **BEGIN** and **END** are also used.

The declarator symbols are: **OWN**, **BOOLEAN**, **INTEGER**, **REAL**, **ARRAY**, **SWITCH**, and **PROCEDURE**.

The specifier symbols are: **STRING**, **LABEL**, and **VALUE**.

The logical value symbols are: **TRUE** and **FALSE**.

A special symbol available with the B 5000 is (?). This symbol is printed whenever an illegitimate character code is encountered.

### IDENTIFIERS

Identifiers serve as names of important entities of a

program. An identifier is composed of letters and digits and must have a letter as its first character. For example:

```
W9XBY
ABLE
B47
ENTRYPOINT
```

There are six types of identifiers: variable, procedure, switch, array, label, and formal parameter.

**Variable.** A variable identifier is a name given to a single arbitrary quantity. The name allows the quantity to be referenced. The quantity may be changed throughout the program.

**Procedure.** A procedure identifier is a name given to a closed and self-contained process with a fixed set of arguments; for example, a subroutine. The name allows this procedure body to be referenced.

**Switch.** A switch identifier is a name given to a program switch. A program switch allows the programmer to make a transfer to one of several statement labels.

**Array.** An array identifier is a name given to an ordered set of values: the variables of a multidimensional array. Any one or all of these values may be changed throughout the program.

**Label.** A label is a name given to a particular statement, so that it may be referred to in other statements.

**Formal Parameter.** A formal parameter is a name given to an argument of a procedure.

## NUMBERS

A number is composed of digits, with additional explanatory symbols allowed in order to denote the sign of the number, decimal point, the existence of a power of ten factor, the sign of the power of ten, and the actual power. For example:

```
-316
+36.001
16 × 10*1
13.62 × 10* -3
-7.5 × 10* -9
```

## STRINGS

A string is a combination of any basic symbols (not containing ") which are bounded by the bracket symbol ". For example:

```
"/ -1AB;("."
"JAMES"
"1268V896"
```

## EXPRESSIONS

The two levels of language presented thus far, the alphabet and words of ALGOL 60, are used in various combinations to form the next language level, expressions. Before considering the three major expressions—arithmetic, Boolean, and designational—the minor expressions of ALGOL 60, variables and function designators, must be discussed. These minor expressions, along with operators, delimiters and reserved words, constitute the major expressions.

## VARIABLES

A variable is a designation given to a single value. This value may be changed at will by means of a particular type of statement in the program. A variable may be either a variable identifier or an array identifier with subscripts. For example:

```
A[1,2]
B[I,J]
EPSILON
M2[X + Y, J + K]
X
```

Since vertical alignment of characters has no meaning in this language, subscripts are enclosed by brackets and separated by commas. These subscripted variables designate values which are components of multidimensional arrays; the subscripts designate which value. The subscripts of a list may be variables, function designators, or arithmetic expressions.

## FUNCTION DESIGNATORS

Function designators are also used for denoting single values, but with the distinction that these values are the result of a specific set of computations on given parameters. A function designator consists of a procedure identifier, which refers to the procedure body containing the specific set of computations, and an actual parameter part, which contains the list of parameters to be used in the computations. For example:

```
SIN(X)
LN(2 × A - 3 × B)
P(X, Y - Z)
```

The parameters of the actual parameter part are enclosed in parentheses and separated by commas. The standard functions of analysis and their procedure identifiers are an integral part of the B 5000 programming language; the procedures to which these identifiers refer need not be defined by the programmer in a procedure body. For example:

SIN  
ARCTAN  
EXP  
SQRT  
ENTIER

The function ENTIER is used for transferring an expression of real type to one of integer type.

### ARITHMETIC EXPRESSIONS

There are two types of arithmetic expressions: simple and general.

**Simple.** A simple arithmetic expression is a description of an algebraic process which produces a numerical value. It is composed of numbers, variables, function designators, and arithmetic operator symbols. For example:

$16.2 \times \text{SIN}(X + Y)$   
 $\text{LN}(2 \times A - 3 \times B) / (-.165 * C \times \text{EPSILON})$   
 $A[1,2] \times M2[X + Y, J - K]$

When numbers are used for operands, the values of the operands are self-evident. When the operands are variables, the numerical values to be operated on are those currently assigned to the variables by a program statement. When a function designator is involved, the value is derived from the procedure body containing the computations which define the function.

**General.** A general arithmetic expression also produces a numerical value. It contains several simple arithmetic expressions and a means for selecting the one which is to be evaluated.

Three sequential operator symbols, Boolean expressions (below) and simple arithmetic expressions are combined as follows:

**IF** (Boolean expression) **THEN** (simple arithmetic expression) **ELSE** (arithmetic expression). For example:

**IF X DIV Y > 0 THEN X + Y ELSE Z**  
**IF A < B THEN B/A ELSE IF A > B THEN**  
**A/B ELSE 1**

The selection is based on the values (**TRUE** or **FALSE**) of the Boolean expressions. The Boolean expressions of the **IF** clauses are evaluated in sequence from left to right until one having the value **TRUE** is found. The simple arithmetic expression then selected for the value of the general arithmetic expression is the one following the next **THEN**.

If none of the Boolean expressions is **TRUE**, the simple arithmetic expression evaluated is the last one in the general arithmetic expression; that is, the one following the last **ELSE**.

In the second example listed above, there is a choice of three simple arithmetic expressions:  $B/A$ ,  $A/B$ , and 1. If, in an actual problem,  $A = 10$  and  $B = 5$ , the second Boolean expression,  $A > B$  would be true and  $A/B$  or 2 would be the numerical value of the expression.

### BOOLEAN EXPRESSIONS

There are also simple and general Boolean expressions.

**Simple.** A simple Boolean expression is a description of a process for computing a logical value: **TRUE** or **FALSE**. The expression is composed of logical value symbols, variables (Boolean type), function designators (Boolean type), logical operator symbols and relations (arithmetic expressions connected by a relational operator). For example:

$A \wedge B$   
 $S \vee Q \text{ EQV } R$   
 $(X - Y) \times Z \leq 5 \times P$

Whenever variables or function designators are used in Boolean expressions, they must be declared Boolean by a special declaration in the program.

Whatever is located on either side of a logical operator symbol must have a logical value. The elements separated by a relational operator symbol must have a numerical value.

Relational operator symbols have familiar meanings, but logical operators are not so commonly known. Their meaning is given in the following chart:

<b>IF</b>	
A	is FALSE FALSE TRUE TRUE
<b>AND B</b>	is FALSE TRUE FALSE TRUE
<b>THEN</b>	
$\neg B$	is TRUE FALSE TRUE FALSE
$A \wedge B$	is FALSE FALSE FALSE TRUE
$A \vee B$	is FALSE TRUE TRUE TRUE
$A \text{ IMP } B$	is TRUE TRUE FALSE TRUE
$A \text{ EQV } B$	is TRUE FALSE FALSE TRUE

**General.** A general Boolean expression is also a description of a process for computing a logical value. It contains several simple Boolean expressions and a means for selecting the one bearing the value to be used. The selection method is analogous to that used for general arithmetic expressions. A general Boolean expression is formulated as follows: **IF** (Boolean expression) **THEN** (simple Boolean expression) **ELSE** (Boolean expression). For example:

**IF L ≥ M THEN K = C ELSE N < Y**  
**IF JOE THEN FALSE ELSE IF PETE THEN**  
**TRUE ELSE JOE**

## DESIGNATIONAL EXPRESSIONS

A designational expression is a rule by which a label of some statement is selected for the purpose of referencing that statement. A simple designational expression is either a label or a switch designator.

Labels have been discussed. A switch designator is used to specify the program path to be followed through a switch, which is a conditional transfer device with a number of choices. The switch designator is composed of a switch identifier followed by a subscript expression enclosed in brackets. In order for the switch designator to have meaning, its arithmetic expression part must assume a positive integral value not greater than the number of choices available in the switch. The switch choices are listed in a declaration to which the switch designator refers. The transfer choice selected is the *n*th label in the declaration switch list counting from left to right, *n* being the integral value of the arithmetic expression part of the switch designator.

For example:

```
A75
MABEL
PICK[N + 1]
```

There is also a general designational expression. Its formulation and evaluation principles are entirely analogous to those of arithmetic expressions. For example:

```
IF A < C THEN A75 ELSE PICK [N + 1]
```

## STATEMENTS

Statements are the programming units of the language; they are comprised of the basic symbols, words, and expressions previously discussed. Statements, like verbal sentences, are complete units of communication. Just as sentences can be combined to form paragraphs, so can basic statements be combined to form compound statements.

For referencing purposes, any basic statement can be given a label. So can a compound statement or a block be labeled. A compound statement might appear as follows: label: **BEGIN** statement; statement; statement;...statement; statement **END**. A block might appear as follows: label: **BEGIN** declaration; declaration;...declaration; statement; statement;...statement; statement **END**

Every block automatically introduces a new level of nomenclature; that is, an identifier occurring within a block may, through a declaration, be specified to be local to the block.

Declarations serve to define certain properties of the identifiers of the program. For example, a procedure

declaration defines the procedure associated with a procedure identifier. Since it is necessary to refer to declarations during the descriptions of the statement types, the brief description of them given above was imperative at this point.

There are six types of statements possible in an ALGOL 60 program.

## ASSIGNMENT STATEMENTS

An assignment statement serves to give a specific value to one or several variables. Each entry in the statement is separated by the symbol  $\leftarrow$ , with the variables listed to the left and the expression by which they are to be replaced at the right. For example:

```
P  $\leftarrow$  P + 1
M2[X + Y, Z]  $\leftarrow$  A [1, 2]  $\leftarrow$  SIN(C - D)
OUT  $\leftarrow$  TRUE
IN  $\leftarrow$  Q ^ R
```

As used in assignment statements the symbol  $\leftarrow$  means "Using the current value of the variables, evaluate the expression. The result is then assigned to all of the variables to the left of the symbol ( $\leftarrow$ )". All variables on the left must be of the same type. If the variables are Boolean, the expression must likewise be Boolean. If the variables are of type real or integer, the expression must be arithmetic.

## GO TO STATEMENTS

A go to statement comprises the sequential operator **GO TO** and a designational expression. For example:

```
GO TO MABEL
GO TO PICK [N + 1]
GO TO IF A < C THEN A75 ELSE PICK [N + 1]
```

A go to statement is used for transfer of control. It serves to interrupt the normal sequence of operations from one statement to the next. The label of the next statement to be executed after a go to statement is defined by the value of the go to statement designational expression.

## DUMMY STATEMENTS

A dummy statement executes no operation. For example:

```
BACK:
BEGIN...; JOHN: END
```

The primary function of a dummy statement is to place a label at the end of a procedure so that transfer of control can be made midway through the procedure back to the place in the program which called for its execution.

## CONDITIONAL STATEMENTS

A conditional statement can cause certain statements to be executed or skipped depending on the values of specified Boolean expressions. A conditional statement in general takes the following form: **IF** Boolean expression **THEN** unconditional compound statement. The **ELSE** portion is arbitrary.

For example:

```
IF A>B THEN X←X +1
IF X=0 THEN X←1 ELSE IF X=9 THEN
GO TO BACK
IF E +F=6 THEN G←1 ELSE GO TO OUT
```

If the Boolean expression of the **IF** clause is **FALSE**, the statement following it will be skipped and operation will continue to the next statement encountered, either the statement following the sequential operator **ELSE** or the next statement in sequence. If the Boolean expression of the **IF** clause is **TRUE**, the statement following it is executed and operation then continues with the next statement in sequence beyond the conditional statement.

## FOR STATEMENTS

A **FOR** statement provides the ability to perform repetitive operations on a statement which is a part of it. The basic structure of a **FOR** statement is as follows: **FOR** variable ← list **DO** compound statement.

A **FOR** statement causes the statement following the **DO** to be repeatedly executed zero or more times with a new assignment to its controlled variable each time. The list provides a rule for obtaining the values which are assigned. The sequence of values is obtained from the list elements (arithmetic expression, **STEP-UNTIL**, or **WHILE**) by taking these one by one in the order in which they are written.

**Arithmetic Expression Element.** The simple case is a single arithmetic expression, the value of which is calculated immediately before the single execution of the statement.

**Step-Until Element.** This element takes the form A **STEP** B **UNTIL** C, where A, B, and C are arithmetic expressions. This element causes the assignment of the current values of A, A + B, A + 2 × B, etc., to the variable and the corresponding execution of the statement following **DO**. The operation terminates (list exhausted) when V (variable) - C is nonzero and has the same sign as B. This test is made just prior to each execution of the statement following **DO**, so that the statement is never executed if initially A - C and B have equal signs, if both are nonzero.

**While Element.** This element takes the form E **WHILE** F, where E is an arithmetic and F a Boolean expression. Each time the variable is assigned the value of the arithmetic expression, a test is made on the Boolean expression. If it has a value of **FALSE**, the list is exhausted. If it is **TRUE**, the statement following **DO** is executed.

## PROCEDURE STATEMENTS

A procedure statement calls for the execution of a procedure declaration body. It is composed of a procedure identifier and an actual parameter list enclosed in parentheses and separated by commas. For example:

```
TRANSPOSE (W,V +1)
ABSMAX (A,N,M,Y,I,K)
```

The number of actual parameters listed must agree with the number of formal parameters given in the procedure declaration heading.

## DECLARATIONS

Declarations define certain properties of identifiers of a program. A declaration for an identifier is valid for one block. Outside this block the particular identifier may be used for other purposes.

A declaration may be marked with the declarator symbol **OWN**. Upon re-entry into a block, values of **OWN** quantities will be unchanged from their values at the last exit; others are undefined.

## TYPE DECLARATIONS

Values of simple variables may be of three types: **INTEGER**, **REAL** or **BOOLEAN**. Type declarations place simple variables into one of these three classes. A type declaration is composed of a declarator symbol followed by all simple variables of that class. The simple variables are separated by commas. For example:

```
REAL W9XBY, X, EPSILON
INTEGER ABLE, B47, ENTRYPOINT
BOOLEAN Y, Z, BAKER
```

Simple variables declared as **REAL** may assume positive and negative values including zero. Those declared as **INTEGER** may assume only positive and negative integral values including zero.

The constituents of a simple arithmetic expression may be either of type **REAL** or type **INTEGER**. In general, if there is a mixture of types within a simple arithmetic expression, the resulting value will be **REAL**. If all constituents are **INTEGER**, the resulting value is **INTEGER**.



## ARRAY DECLARATIONS

An **ARRAY** declaration specifies one or more identifiers to represent multidimensional arrays and gives the array dimensions, the upper and lower bounds of the subscripts, and the types of the variables. An **ARRAY** declaration comprises declarator symbols, array identifiers, and bound pair lists. For example:

```
INTEGER ARRAY A[1:6, 1:3], B[-N:O, O:N]
ARRAY M2 [IF X<0 THEN 0 ELSE 1:6],
MA[C:D]
```

A bound pair is two arithmetic expressions separated by the symbol (:). The first expression is the lower bound and the second the upper bound of a subscript. A bound pair list is a series of bound pairs separated by commas and giving the bounds of all subscripts of an array taken in order from left to right. The bound pair list immediately follows the identifier to which it pertains.

The dimension of an array is given as the number of entries in the bound pair list.

All arrays specified in one declaration are of the same quoted type. If no type is quoted in an **ARRAY** declaration, type **REAL** is understood.

## SWITCH DECLARATIONS

A switch declaration lists all program transfer choices available at any one switch. It is composed of the symbol **SWITCH**, a switch identifier, the symbol ( $\leftarrow$ ) and a list of designational expressions separated by commas. For example:

```
SWITCH PICK $\leftarrow$ P1, P2, P1 + 6
SWITCH Y $\leftarrow$ A75, MABEL, IF A<C THEN
A75 ELSE PICK[2]
```

A **SWITCH** declaration is referred to each time a switch designator is encountered in the program. The transfer choice taken is the nth expression of the **SWITCH** list, where n is the value of the arithmetic expression associated with the switch designator. The expression selected is then evaluated using the current values of all variables involved.

## PROCEDURE DECLARATIONS

A procedure declaration defines in detail the procedure associated with a procedure identifier. A procedure declaration is made up of a heading and a body.

**Procedure Declaration Heading.** The heading begins with the symbol **PROCEDURE** followed by the procedure identifier. Next comes a formal parameter part which is a list of formal parameters enclosed in parentheses and separated by commas. The formal parameter part is followed by a value part. The value

part starts with the symbol **VALUE** and is followed by the formal parameters which are to be replaced by the values of the actual parameters when the procedure body is executed. The last part of the heading is a specification part which describes each formal parameter. It may specify that it is an **ARRAY**, **STRING**, **PROCEDURE**, **LABEL**, **SWITCH**, etc. Specifier symbols and/or declarator symbols are given followed by the formal parameters which are of that type. A sample procedure declaration heading follows:

```
PROCEDURE POLYROOT (A,X,Q1, TRIG);
VALUE A, X; REAL A, X; ARRAY Q1;
LABEL TRIG
```

**Procedure Declaration Body.** The procedure declaration body comprises a statement or piece of code. The body may be activated from other parts of the block in the head of which the procedure declaration appears by means of procedure statements and/or function designators.

Before execution of the procedure body takes place (this is caused by execution of a procedure statement), all formal parameters quoted in the value part of the procedure declaration are replaced by their corresponding actual parameter values. Formal parameters not quoted in the value list are replaced, throughout the procedure body, by the actual parameters themselves, not the values thereof.

Finally the modified procedure body is inserted in the place of the procedure statement in the program sequence and executed.

## ALGOL EXAMPLE

To give the reader a feeling for this language an example follows. The problem is to find the Gamma Function.

$$Y = \Gamma(z)$$

The method chosen for the calculation is based on the fact that  $\Gamma(1+x)$  can be approximated by an eight degree polynomial of the form:

$$F(x+1) = 1 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7 + a_8x^8$$

where:

$a_1 = - .57719165$	$a_5 = - .75670408$
$a_2 = .98820589$	$a_6 = .48219939$
$a_3 = - .89705694$	$a_7 = - .19352782$
$a_4 = .91820686$	$a_8 = .035868343$

The job is to write a subroutine which can be used by any program which requires the evaluation of a Gamma Function. The range of the variable for

which this approximation of the Gamma Function is defined is:

$$0 < z < 41.15$$

It is assumed here that any program making use of this routine will, before calling for its execution, have determined that the variable is within the defined range.

One of three different formulas applies, depending upon the value of the variable within the defined range. They are:

1. If  $0 < z \leq 1$       $Y = \frac{1}{z} F(1+z)$  is used.
2. If  $1 < z \leq 2$       $Y = \Gamma[1+(z-1)]$  is used.
3. If  $z > 2$           $Y = (z-n)(z-n+1)\dots(z+1)$   
 $F(z-n)$  is used.  
where  $1 \leq z-n < 2$

For  $z > 2$  the integer  $n$  allows for the introduction of a new variable,  $z-n$ , which is then in the second range above. The function  $\Gamma(z-n)$  can then be calculated by means of the second formula. The third formula gives the relationship between  $\Gamma(z)$  and  $\Gamma(z-n)$ .

After making the necessary substitutions, the above three equations become:

1.  $Y = \frac{1}{z} (1 + a_1z + a_2z^2 + a_3z^3 + a_4z^4 + a_5z^5 + a_6z^6 + a_7z^7 + a_8z^8)$
2.  $Y = 1 + a_1(z-1) + a_2(z-1)^2 + a_3(z-1)^3 + a_4(z-1)^4 + a_5(z-1)^5 + a_6(z-1)^6 + a_7(z-1)^7 + a_8(z-1)^8$
3.  $Y = (z-n)(z-n+1)\dots(z+1) \times [1 + a_1(z-n-1) + a_2(z-n-1)^2 + a_3(z-n-1)^3 + a_4(z-n-1)^4 + a_5(z-n-1)^5 + a_6(z-n-1)^6 + a_7(z-n-1)^7 + a_8(z-n-1)^8]$

where  $n = \text{ENTIER}(z-1)$

ENTIER is a function which produces an integral value from an expression which may be composed of nonintegral values. For instance,  $n = \text{ENTIER}(z-1)$  means that if  $z = 4.5$ , then  $n = 3$  or the largest integer which is not greater than the function argument  $(z-1)$ .

It can be noted here that ENTIER is a standard function in the B 5000 language and is automatically available for use by the programmer, as are others, such as SIN, ABS, SQRT, ARCTAN, LN, etc.

A solution of this problem using the ALGOL part of the B 5000 programming language is given below. The keypunch operator would punch exactly that

which is shown.

```
PROCEDURE GAMMA (Z,Y); VALUE Z;
REAL Z,Y; INTEGER M, N;
```

```
  BEGIN
```

```
    REAL PROCEDURE POLYGAM (X);
```

```
    VALUE X; REAL X;
```

```
    BEGIN
```

```
      REAL Y;
```

```
      Y ← 1 - .57719165 × X + .98820589
        × X*2 - .89705694 × X*3 + .91820686
        × X*4 - .75670408 × X*5
        + .48219939 × X*6 - .19352782
        × X*7 + .035868343 × X*8;
      POLYGAM ← Y
```

```
    END
```

```
  COMMENT: START OF GAMMA PRO-
  CEDURE PROGRAM;
```

```
  IF Z < 1 THEN Y ← POLYGAM (Z)/Z
```

```
  ELSE IF Z ≤ 2 THEN Y ← POLYGAM
  (Z - 1)
```

```
  ELSE BEGIN
```

```
    M ← ENTIER (Z - 1); Y ← POLYGAM
    (Z - M - 1);
```

```
    FOR N ← M STEP -1 UNTIL -1
```

```
      DO Y ← Y × (Z - N)
```

```
  END
```

```
END
```

NOTE

Even in a problem-oriented language the programmer can make use of his ingenuity by formulating expressions in ways that decrease the running time of object programs. For example, the first assignment statement of the procedure (POLYGAM) could have been written as  $Y \leftarrow ((((((((.035868343 \times X - .19352782) \times X + .48219939) \times X - .75670408) \times X + .91820686) \times X - .89705694) \times X + .98820589) \times X - .57719165) \times X + 1$

A solution to the same problem written in the machine language of another computer has been formulated as follows. Again, only what is to be punched by the keypunch operator is shown.

00000047002	00000000000
00000080020	05110000000
00000737002	00000807048
00000127019	00000127049
20000020038	20000020069
00000647001	00000647047
00000140004	00000837049
20000020056	20000020059
00000110000	20000300050
20000020057	05110000000
00000647019	05110000000
00000817018	00000000000
00000737012	00000727058
20000380040	00000827069
00000757018	10000807060
20000300020	00000227051
00000000000	00000827059
00000000000	00000727057
05110000000	00000000000
00000000000	00000000000
00000737020	00000000008
20000380070	00000000000
00000647038	05110000000
00000817036	15057719165
00000127038	05098820589
00000827037	15089705694
00000127037	05091820686
00000647038	15075670408
00000807035	05048219939
00000737035	15019352782
00000287022	04935868343
00000807039	00000000000
20000020069	00000747075
00000647037	20000020069
20000300073	00000647075
15120000000	20000020059
05110000000	20000300050
05110000000	05110000000

Even though it is a challenging and to some an interesting exercise to produce an actual machine language program, it seems obvious from the above examples that if one were interested in getting an answer as fast as possible, he would choose the former method.

In order to establish an association between the syntactic rules of ALGOL and the ways in which those rules are used in writing a program, the previous solution is presented (Pg.9) with individual parts identified. All English words which serve as basic symbols of the language are printed boldface.

The question may arise as to how this procedure declaration is fitted into another program. Examination of the construction of an over-all program yields the answer.

Sections of a program are called blocks or compound statements. Both are bounded by the bracket symbols **BEGIN** and **END**. It is interesting to note that the creators of ALGOL could have chosen single characters for this purpose such as (&). Instead, English words were chosen because of their familiar meaning. The only difference between a block and a compound statement is that a block contains declarations immediately after the left bracket symbol, **BEGIN**, and compound statements do not. It can be seen then that the declaration outlined above for evaluating a Gamma Function must appear in the head of some block of the program that uses it.

In order to initiate its use, the program must also contain a procedure statement of the form:

	Procedure	Actual
	Identifier	Parameters
Procedure		
Statement	GAMMA	(BETA, MU)

Writing the above statement in the program would result in MU assuming the value of  $\Gamma$  (Beta).

PROCEDURE  
PROCEDURE  
PROCEDURE HEADING  
PROCEDURE DECLARATION  
PROCEDURE BODY

DECLARATOR SYMBOL PROCEDURE IDENTIFIER FORMAL PARAMETER PART  
**PROCEDURE** GAMMA (Z, Y);

SPECIFICATOR SYMBOL  
**VALUE** Z;

VALUE PART

DECLARATOR SYMBOL IDENTIFIER LIST  
**REAL** Z, Y; **INTEGER** M, N;

SPECIFICATION PART

BRACKET SYMBOL  
**BEGIN**

FORMAL PARAMETER  
**REAL PROCEDURE** POLYGAM (X);

SEPARATOR SYMBOL  
**VALUE** X;

VARIABLE IDENTIFIER  
**REAL** X;

**BEGIN**

TYPE DECLARATION  
**REAL** Y;

ASSIGNMENT STATEMENT  

$$Y \leftarrow 1 - .57719165 \times X + .98820589 \times X^2 - .89705694 \times X^3 + .91820686 \times X^4 - .75670408 \times X^5 + .48219939 \times X^6 - .19352782 \times X^7 + .035868343 \times X^8;$$

IDENTIFIER VARIABLE  
**POLYGAM** ← Y

**END**

BLOCK HEAD  
BLOCK  
BLOCK HEAD  
COMPOUND TAIL

BLOCK HEAD  
BLOCK  
BLOCK HEAD  
COMPOUND TAIL

DECLARATION

BODY

SEPARATOR SYMBOL      LETTER  
 ┌───┴───┘                    ┌──┴──┘  
**COMMENT: START OF GAMMA PROCEDURE PROGRAM;**

IF STATEMENT

SIMPLE BOOLEAN EXPRESSION    SEQUENTIAL OPERATOR SYMBOL    SIMPLE ARITHMETIC EXPRESSION  
 ┌──┴──┘    ┌──┴──┘    ┌──┴──┘  
**IF Z < 1 THEN Y ← POLYGAM (Z) /Z**

RELATIONAL OPERATOR SYMBOL    PROCEDURE IDENTIFIER    ACTUAL PARAMETER LIST  
 ┌──┴──┘    ┌──┴──┘    ┌──┴──┘  
**ELSE IF Z ≤ 2 THEN Y ← POLYGAM (Z -1)**

**ELSE**

**BEGIN**

CONDITIONAL STATEMENT

SEPARATOR SYMBOL    STANDARD FUNCTION DESIGNATOR    FUNCTION DESIGNATOR  
 ┌──┴──┘    ┌──┴──┘    ┌──┴──┘  
**M ← ENTIER (Z -1); Y ← POLYGAM (Z -M -1);**

FOR STATEMENT

STEP-UNTIL ELEMENT  
 ┌──┴──┘    ┌──┴──┘  
**FOR N ← M STEP -1 UNTIL -1 DO Y ← Y × (Z -N)**

**END**

**END**

# APPENDIX B

## B 5000 DATA-PROCESSING LANGUAGE

In support of the Conference of Data Systems Languages (CODASYL), the data-processing language for the B 5000 incorporates the latest revision to the Common Business Oriented Language Specifications, COBOL 61.

### DEVELOPMENT

The Initial Specifications for the Common Business Oriented Language were published in May, 1960, by the Government Printing Office and are currently available from the Superintendent of Documents, Washington 25, D.C. The Executive Committee defined the Initial Specifications as COBOL 60.

The COBOL Maintenance Committee then began the arduous process of removing the obvious editing errors, the ambiguities, and the inconsistencies which they as a committee discovered and agreed upon. The current revision is due to be "frozen" for republication as COBOL 61 on February 4, 1961.

The BURROUGHS Data-Processing Language for the B 5000 will allow source programs to be written according to the COBOL 61 Specifications. In order that the integrity and continuity of COBOL can be maintained for the using public, the B 5000 Programming System provides the ability to incorporate future revisions such as COBOL 62.

### EXTENSIONS

As necessary extensions to the COBOL 61 Specifications, the B 5000 Programming System provides the following standard set of commercial functions:

1. Punched card input/output conversion.
2. Formating, editing, and validity checking of important data.
3. Master file creation.
4. Master file maintenance.
5. Generalized sorting.
6. Operational data error control.
7. Efficient program segmentation.
8. Automatic input/output unit assignment.
9. Automatic tape label and file control.
10. Management exception report creation.

11. COBOL source-language debugging techniques.
12. Automatic entry to and return from the computational language (ALGOL).

### ADVANTAGES

The B 5000 Data-Processing Language offers the following advantages:

1. Powerful and flexible means of accurately defining business problems.
2. A major reduction of the time between problem definition and full applicational production.
3. Flexibility for management to revise its current methods of doing business (Management Control System) by updating automatic data processing (ADP) programs without the excessive delays and exorbitant reprogramming costs associated with present-day automatic data processing systems (ADPS).
4. For those customers requiring compatibility between ADP systems, the B 5000 Data Processing Language provides all those elements of COBOL 61 which the Conference of Data Systems Languages (CODASYL) agreed were "required" to achieve a common implementation across all participating manufacturers' ADPS.
5. The opportunity for the B 5000 Users Group to participate in the formation of a Business Data Processing Source-Language Library of Procedures (subroutines) and to exchange problems in documented procedural source-language form.
6. An "open-ended" programming system having the ability to incorporate future improvements, for example, COBOL 62.

### GENERAL DESCRIPTION

A source program is used to specify the solution of a business data processing problem. The four elements of this specification are:

1. The identification of the source program.
2. The description of the problem environment, that is, the ADPS to be used to compile the source program and to operate the object

program.

3. The description of the data to be processed.
4. The set of procedures which determine how the data is to be processed.

COBOL 61 has a separate division for each of these elements. The names of these divisions are: Identification, Environment, Data, and Procedure.

COBOL 61 allows the user to prepare his specifications for the problem solution in the language most natural to him—English.

The Identification Division provides a means to identify or label a COBOL source program.

The Environment Division is that part of the source program which specifies the equipment being used. It contains descriptions of the computers to be used both for compiling the source program and running the object program. Memory size, number of tape units, hardware switches, printers, etc., are among many items that may be mentioned for a particular computer.

The Data Division uses file and record descriptions to describe the files of data and the individual logical records which comprise these files that the object program is to create or manipulate.

The Procedure Division specifies the steps that the user wishes the computer to follow. These steps are expressed in terms of meaningful English words, statements, sentences, and paragraphs. This aspect of the over-all system is often referred to as the “program.” In reality it is only part of the total specification of the problem solution and is insufficient to describe the entire problem. This is true because repeated references must be made—either explicitly or implicitly—to information appearing in the other divisions.

The Procedure Division—more than any other—allows the user to express his thoughts in meaningful English. Concepts of verbs to denote actions, and sentences to describe procedures, are basic, as is the use of conditional statements to provide alternative paths of action.

In order to provide a standard method of writing COBOL 61 source programs, the CODASYL selected the following Reference Format:

LEFT MARGIN					RIGHT MARGIN
L		A		B	R
SEQUENCE NUMBER					
(6)	(1)	(4)			

NOTE: (n) = number of spaces, A denotes 8th column and B the 12th column

The standard method of representing sentences, paragraphs, and sections, is shown below for each division. The principle behind all the formats chosen is to allow the maximum amount of flexibility for individual tastes while still using one form.

The four divisions of a COBOL 61 program must appear in the following order: IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE.

## IDENTIFICATION DIVISION

The Identification Division employs the Reference Format in the following way.

Each paragraph name starts under position A. The text of each paragraph may start anywhere on the same line as the paragraph name or on the next line starting under position B. (See Example 1).

## ENVIRONMENT DIVISION

The Reference Format is employed in the Environment Division in the same way as in the Identification Division. In addition to fixed paragraph names, there are also fixed section names. The rules for continuation of sentences and words are the same as those in the Procedure Division. The Input-output control and the File-control paragraphs are each composed of several sentences, whereas the other paragraphs are each composed of one sentence only. Example 2 shows a possible Environment Division, except that all necessary information for each section and each paragraph has not been shown.

## DATA DIVISION

The basic unit in the Data Division is an entry. Each entry begins with a level number followed by the name of the datum and a sequence of independent clauses descriptive of the datum. Each clause of an entry (except the last) may be terminated by a semicolon followed by a space. The last clause is terminated by a period. The same format rules apply to the File and Record Description portions as well as the Working Storage and Constant Sections of the Data Division.

The sequence number appears at the left as called for in the Reference Format. The first-level number starts under position A. If a single entry requires more than one printed line, the left margin for each line is the same, namely, the position under the first letter of the data name. The rules for continuation of words are the same as those in the Procedure Division. (See Example 3).

### EXAMPLE 1

SEQUENCE NO.	L	A	B	C	D	R
1		78	12	16	20	120
000100		IDENTIFICATION-DIVISION.				
000200		PROGRAM-ID. UPDATE-MASTER-ACCTS-RECEIVABLE.				
000300		AUTHOR. LARRY STURGESS.				
000400		DATE-WRITTEN. JANUARY 12, 1961.				
000401		REMARKS. RUN 046660. INPUT FROM CASH-RECEIPTS. RUN 045660.				

### EXAMPLE 2

SEQUENCE NO.	L	A	B	C	D	R
1		78	12	16	20	120
001100		ENVIRONMENT-DIVISION.				
001200		CONFIGURATION SECTION.				
001300		SOURCE-COMPUTER. STANDARD-B5000.				
001301		OBJECT-COMPUTER. PARALLEL-COMMERCIAL-B5000.				
001302		INPUT-OUTPUT SECTION.				
001303		FILE-CONTROL SELECT MASTER-ACCT-RECEIVABLE, FOR MULTIPLE REEL, PRIORITY IS 053.				
001304		SELECT CASH-RECEIPTS, PRIORITY IS 050.				
001400		I-O-CONTROL. APPLY PARALLEL-INPUT-OUTPUT. RERUN EVERY CHECK-POINT.				

## PROCEDURE DIVISION

The first line of the Procedure Division consists of its name starting under position A followed by a period:

PROCEDURE-DIVISION.

If a section has been designated, the section name starts under position A, followed by a space, the word SECTION, a space followed by the priority number (when used), a period, and a space. A section name may be used as a qualifier for otherwise identical paragraph names. Within a section, any reference to a paragraph name not otherwise qualified will be assumed to refer to paragraphs within the section. A paragraph consists of one or more successive sentences, the first and only the first of which must be named. The paragraph assumes the name given to the first sentence. The name starts under position A and is followed immediately by a period and a space. The first sentence of the paragraph may begin anywhere on the same line as the paragraph name, or under position B on the next line. A new paragraph is determined by the appearance of another para-

graph name. Note that a paragraph may possibly consist of a single sentence.

Any sentence which occupies more than one line must be continued by starting under position B on the next line.

If a word or literal must be split over two lines, this will be indicated by placing a hyphen in the seventh character position on the next line, that is the character position between the least significant digit of the sequence number and the first character of the procedure name. If the user prefers not to split a word or literal, he may start the word or literal on the next line. (See Example 4.)

## CHARACTER SET

### Characters Used in Forming Words

Groups of characters selected from the following 37 characters are called "words."

- 0, 1, ... , 9
- ✓ A, B, ... , Z
- (hyphen or minus sign)



### EXAMPLE 3

SEQUENCE NO.	L	A	B	C	D	R
1		78	12	16	20	120
003100						
003202						
003203						
003204			01			
003205				02		
003206				02		
003301				03		
003302				03		
003306				03		
003307				02		
003401			01			
003402				02		
003403						
003404				02		

### EXAMPLE 4

SEQUENCE NO.	L	A	B	C	D	R
1		78	12	16	20	120
010100						
010101						
010102						
010103						
010104						
010105						
010106						
010107						
010108						

A "blank" or "space" is not an allowable character for a word, but is used to separate words and statements. Where a "blank" or "space" is employed, more than one may be used, except in the Reference Format.

) right parenthesis  
 space  
 . period  
 , comma  
 ; semicolon

#### Characters Used for Punctuation

The punctuation characters consist of the following:

" quotation marks  
 ( left parenthesis

#### Characters Used in Relations

> greater than  
 < less than  
 = equal to

(These characters are not in the standard character set used to write COBOL 61 programs. They are available for optional use as extensions to the B 5000 Data Processing Language.)

### Characters Used in Editing

\$	dollar sign
*	check protection symbol
,	comma
.	decimal point

### WORDS

A word is composed of not more than 30 characters chosen from the set of 37 characters used for words as listed above. A word is ended by a space, or a word is ended by either a period, right parenthesis, comma, semicolon, followed by a space. All of the word types defined below (except the literal) may not begin or end with a hyphen.

The use of punctuation characters in connection with words is as follows: A period, comma, and semicolon, when used, must always immediately follow a word, and they must in turn be followed by a space. A left parenthesis or a beginning quote mark (see Literals) must not be followed by a space unless the space is desired in the data-name or literal. A right parenthesis or ending quote mark must not be preceded by a space unless the space is desired in the data-name or literal.

For example:

- "NL5960" represents a literal which has a SIZE of 6 characters and the VALUE of NL5960.
- ( 10) represents a table entry equivalent in SIZE but not in VALUE to
- (010)

### Nouns

A noun is a single word used for some form of referencing. Three examples of nouns are:

- ✓ Data-name
- ✓ Procedure-name
- ✓ Literal

A Data-name is a noun which contains at least one alphabetic character and which is used for describing data. A Data-name is either a File-name, a Record-name, a Group-name, or an Elementary-Item-name.

Procedure-names are used so that one procedure can reference another. A Procedure-name is either a Sentence-name, a Paragraph-name, or a Section-name. A Procedure-name is a noun which may be

composed of purely numeric characters. Two numeric Procedure-names are equivalent only if they are composed of the same number of numeric characters, and have the same numeric value.

A literal is a noun that represents a value whose length may not be composed of more than 120 characters. A literal is either numeric or non-numeric. Non-numeric literals must be bounded by quotation marks. A non-numeric literal may not contain a quotation mark within itself.

For example:

- "1234A56" is a non-numeric literal whose size is 7 characters, whereas
- "123"56" represents an illegal entry. The Programming System might select the literal, "123", and assign a SIZE of 3 characters, and note an error as a result of compilation.

A numeric literal is defined as one which is composed only of characters chosen from the numerals 0 through 9, the plus (+) or minus (-) sign, and the decimal point. The rules for formation of numeric literals are:

1. A numeric literal must contain only one sign character and/or one decimal point.
2. The literal must contain at least one digit.
3. The sign in the literal must appear as the leftmost character of the literal. If the literal is unsigned, the literal is considered as positive.
4. The decimal point in the literal may appear anywhere within the literal except as the rightmost character of the literal. If the literal contains no decimal point, the literal is considered to be of *integer* value.

If a literal conforms to the rules for formation of numeric literals but is enclosed in quotation marks, it is considered as a *non-numeric* literal and will be treated as such by the compiler.

For example:

- +646.90 is not equivalent to
- "+646.90"

### Qualification

Qualification is used to differentiate between like names which appear in two or more places. Qualification is performed by appending one or more preposition phrases using the preposition IN or OF. The choice between IN and OF is based only on readability because they are logically equivalent. Nouns must appear in ascending order of hierarchy with either of the words IN or OF separating them. The qualifiers are considered part of the name. Thus

whenever a value is referenced, the qualifiers are automatically considered part of the name.

For example, consider two files named:

MASTER and  
NEW-MASTER.

Assume that each of them contains a Data-group named

CURRENT-DATE, and another named  
LAST-TRANSACTION-DATE.

If both of these Data-groups contain three Elementary-Items with Data-names Month, Day, and Year, then the current month in the NEW-MASTER record is referred to as:

MONTH IN CURRENT-DATE OF NEW-MASTER, and

The Day of the last transaction in the MASTER is referred to as:

DAY IN LAST-TRANSACTION-DATE  
OF MASTER

### Subscripts

In business data processing a commonly used operation is that of "table lookup." COBOL 61 provides the ability to reference individual elementary-data-elements of a table or a list by using subscripts. Additionally, the ability to reference the *entire* table or list is provided by referencing the name of the table or list.

A subscript is an integer whose value determines which elementary-data-element within a table (or a list) is to be referenced. The subscript itself is represented by either an integer literal, e.g. (25), or by a Data-name which has an integer value, e.g. (AGE).

In all cases, the subscript is enclosed in parentheses and appears immediately after the terminal space of the name of the data element being referenced, e.g. RATE (AGE) or RATE (25). Tables are often defined so that more than one level of subscripting is required to locate an element within the table. A maximum of three dimensions is permitted by COBOL 61. Multi-level subscripts are always written from left to right in the order major, intermediate, and minor. In this case the subscripts are shown in one pair of parentheses and separated by commas. For example, RATE (REGION, STATE, CITY) would reference a particular rate in a 3-dimensional table of rates.

A subscript with integer value of "1" denotes the first elementary-data-element of a list, a value of "2" the second elementary-data-element, etc. A subscript of (1,2) represents the second elementary-data-element

within the first repeated elementary-data-element in the table. A table with its main element appearing 10 times, its intermediate element appearing 5 times within each of the major elements, and the minor elementary-data-element appearing 3 times within each intermediate element is considered 3 dimensional. The last element of such a table is referenced by the subscript (10, 5, 3).

No element of a table or list may be referenced without a subscript. However, the entire table may be referenced, providing the table has been given a unique name. Reference to a Data-name within a table or list must include all subscripts upon which the Data-name is dependent.

### EXAMPLES:

MOVE RATE (AGE) TO LISTING  
IF HEIGHT (10) IS GREATER THAN ...  
MULTIPLY PRICE (STOCK-NO) BY  
INVENTORY (STOCK-NO)  
EXAMINE CLASS (REGION) REPLACING  
MOVE RATE-TABLE TO OUTPUT-AREA

### NOTATION

The following notation is used in COBOL verb and description formats:

- 1. All upper case words which are underlined are required when the functions of which they are part are used. An error will occur at compilation time if the underlined words are absent or incorrectly spelled.*
- 2. All upper case words which are not underlined are used for readability only. They may be present or absent.*
- 3. All lower case words represent generic quantities whose value must be supplied by the user.*
- 4. Material in braces { } indicates that a choice from the contents must be made.*
- 5. Material in square brackets [ ] represents an option and may be included or omitted at the user's choice.*

### Verbs

The COBOL 61 verbs included in the B 5000 Data Processing Language, listed by categories, are:

Arithmetic	ADD SUBTRACT MULTIPLY DIVIDE
Input-Output	READ WRITE OPEN CLOSE

	ACCEPT DISPLAY
Procedure-Branching	GO ALTER PERFORM
Data-Movement	MOVE EXAMINE
Ending	STOP
Compiler-Directing	DEFINE* ENTER* INCLUDE* USE*

*\*These verbs are not in "required" COBOL 61. They are provided as optional but necessary extensions to accomplish common business data processing operations.*

Sample verb formats together with examples are given below to give the "flavor" of the narrative language.

**ADD** <sup>293</sup> *addition*

FUNCTION: To add two or more quantities and store the sum in either the last named field or the specified one.

**ADD** { literal-1  
data-name-1  
field-name-1 } [ { literal-2  
data-name-2 } ... ]  
[ { TO  
GIVING } ] data-name-n

If the GIVING option is used, the sum of the addends will be stored in "Field-name-n." If the TO option is used, "Field-name-n" is used as both an addend and the result field. If neither option is used, the last named field will be used as both an addend and the result field.

**EXAMPLES:**

1. **ADD** 645 **TO** SUM-OF-PARTS.
2. **ADD** DAY OF RECEIPT-MASTER **TO** DELINQUENT-NOTICE **GIVING** FOLLOW-UP-DATE.
3. **ADD** INITIAL-AMOUNT, SURCHARGE, BONUS **GIVING** TOTAL DUE.

**ALTER**

FUNCTION: To vary the sequence of procedure execution.

**ALTER** procedure-name-1 **TO** { PROCEED } **TO** GO  
procedure-name-2 [ , procedure-name-3

**TO** { PROCEED } **TO** procedure-name-4 ]

"Procedure-name-1," "procedure-name-3," ... are paragraph-names. If a GO statement is to be ALTERed, then the GO statement must be the only statement in the sentence, which in turn must be the only sentence in the paragraph.

**EXAMPLE**

1. **ALTER** FINAL-SUM-TOTAL **TO** GO **TO** CROSSFOOT-TOTAL.

**GO**

FUNCTION: To depart from the normal sequence of procedures.

Option 1:

**GO** **TO** procedure-name

Option 2:

**GO** **TO** procedure-name-1 [ , procedure-name-2... ], procedure-name-out-of-range **DEPENDING** ON data-name

The branch will be to the 1st, ...nth procedure-name as the value of the range of the data-name is 1 to n. If the value of the data-name is out of the positive integral range from 1 to n (i.e. less than 1 or greater than n) the branch will be to "procedure-name-out-of-range."

**EXAMPLES:**

**GO** **TO** MASTER-UPDATE-PROCEDURE.  
**GO** **TO** FIRST-PREMIUM-PROCEDURE, DELINQUENT-PROCEDURE, TERMINATION-PROCEDURE, NOTICE-ERROR PROCEDURE, **DEPENDING** ON STATUS-CODE.

**READ**

FUNCTION: To get the next logical record from an input file; to allow performance of an imperative statement when end of file is detected.

**READ** file-name RECORD [ ; **IF** **END** any imperative statement ]

An OPEN statement for the file must be executed prior to the execution of the first READ for that file.

Every READ statement must have an END of file option, either implicitly or explicitly. The user need only write an END of file option with the first READ for a given file. The compiler will append this and its associated "imperative statement" to each

READ for that file which has no explicit END of file option. If more than one, but not all READs, for the same file contain the word END, the compiler will assign the first one to all other READs for that file which do not have an END option. An error will be indicated as a result of compilation.

EXAMPLES :

READ MASTER-CUSTOMER-ACCOUNT RECORD; AT END GO TO FINAL-TOTAL-PROCEDURE

READ MASTER-CUSTOMER-ACCOUNT.

Note in the last example that the optional word RECORD is missing. Optional words are included as are optional separators to enhance readability. However when optional words are used they must be both correctly spelled and cannot have other optional words substituted for them.

## PROCEDURES

COBOL procedures are expressed in a manner similar to normal English prose. The basic unit of procedure formation is a sentence, which consists of one or more statements. A paragraph is a sentence or a group of successive sentences. A section is a paragraph or a group of successive paragraphs. A procedure therefore is a paragraph, or a group of successive paragraphs, or a section, or a group of successive sections within the Procedure Division.

### Conditionals

Conditional procedures are one of the keystones in describing data-processing problems. COBOL makes available to the programmer a powerful means of expressing conditional situations.

COBOL conditionals involve the key word IF followed by the conditions to be examined followed by the operations to be performed. Depending on the truth or falsity of the conditions different sets of operations are to be performed.

A condition is an expression which has the value "True" (1) or "False" (0).

**SIMPLE CONDITIONS.** A simple condition is either a condition name, a test, or a relation.

**Condition Name.** A field whose specific values can be named is called a conditional variable. The names given to the values are called condition-names. These may be tested to determine whether or not the values as designated are present.

**Tests.** It is possible to determine the status of a field by means of the following tests:

$$\underline{\text{IF}} \left\{ \text{data-name} \right\} \text{ IS } \underline{\text{[NOT]}} \left\{ \begin{array}{l} \underline{\text{POSITIVE}} \\ \underline{\text{NEGATIVE}} \\ \underline{\text{ZERO}} \end{array} \right\}$$

The explicit interpretations of positive, negative, and zero are: Numeric data is POSITIVE only if it is greater than ZERO. Numeric data is NEGATIVE only if it is less than ZERO. A numeric data-element whose value is ZERO is both NOT POSITIVE and NOT NEGATIVE.

If the field has been defined as NON-NUMERIC it may be tested:

$$\underline{\text{IF}} \text{ data-name IS } \underline{\text{[NOT]}} \left\{ \begin{array}{l} \underline{\text{ALPHABETIC}} \\ \underline{\text{INTEGER}} \end{array} \right\}$$

The explicit interpretation of ALPHABETIC and NON-NUMERIC are:

1. A NON-NUMERIC field is ALPHABETIC if it contains any of the 26 letters of the alphabet.
2. If the field contains any of the other characters of the permissible character set for data-fields it is NOT ALPHABETIC. For example, OFFICE is ALPHABETIC, whereas JOHN-DOE is NOT ALPHABETIC.
3. A NON-NUMERIC field is INTEGER if it consists only of the digits 0, 1, 2, ... 9. The presence of any other character renders it NOT INTEGER.

For example, 05462 is INTEGER whereas 054.62 is NOT INTEGER.

**Relations.** A relation is a comparison of two elementary-data-element (fields).

Comparison of a NUMERIC field with a NON-NUMERIC field is a meaningless operation.

For numeric fields, a comparison results in the determination that one of the fields is LESS THAN, EQUAL TO, or GREATER THAN the other.

The comparison of numeric fields is based on the respective values of the fields considered purely as numbers. The field length in terms of the number of digits is not itself significant. Thus, +0042 is GREATER THAN +25.110. Zero is considered to represent a unique value regardless of length of field, sign, or decimal point. Thus, +0.00 is EQUAL TO -0000.

For non-numeric fields, a comparison results in the determination that one of the fields is LESS THAN, EQUAL TO, or GREATER THAN the other with respect to the predefined ordering of the members of the character set.

The comparison of non-numeric fields is based on the assumption that the information contained in the field is left-justified with respect to the left-hand

boundary of the field. There are two cases to consider: fields of equal length and fields of unequal length.

If the fields are of equal length, comparison proceeds by comparing characters in corresponding character positions starting from the left and continuing until either a pair of unequal characters is encountered or the right-hand field boundary is reached, whichever comes first. The fields are determined to be EQUAL when the right-hand boundary is reached.

The first encountered unequal pair of characters is compared for relative location in the ordered character set. The field which contains that character which is positioned higher in the ordered sequence is determined to be the GREATER field.

If the fields are of unequal length, the shorter of the two fields is assumed to be extended by right-concatenating to it a string of whichever character has been defined to be the lowest in the ordered sequence of characters. Having thus made the fields of conceptually equal length the previous case applies. The RELATIONS provided are:

$$\left\{ \begin{array}{l} \text{IF } \left\{ \begin{array}{l} \text{data-name} \\ \text{literal} \end{array} \right\} \\ \text{IS } \left[ \text{NOT} \right] \left\{ \begin{array}{l} \text{GREATER than} \\ \text{LESS than} \\ \text{EQUAL to} \end{array} \right\} \left\{ \begin{array}{l} \text{data-name} \\ \text{literal} \end{array} \right\} \end{array} \right\}$$

The following symbols are equivalent to their Reserved Words and are available as extensions to the standard character set of COBOL 61.

> GREATER than  
 < LESS than  
 = EQUAL to

Examples of RELATIONS are:

1. IF TOTAL > 9999
2. IF STOCK-SIZE = "A54B"

COMPOUND CONDITION. A compound condition is a sequence of simple conditions connected by logical connectives AND, OR, and NOT. The form of a compound condition is:

$$\text{Simple-condition-1 } \left\{ \begin{array}{l} \text{AND} \\ \text{OR} \end{array} \right\} \text{ Simple-condition-2}$$

$$\left\{ \begin{array}{l} \text{AND} \\ \text{OR} \end{array} \right\} \left[ \text{NOT} \right] \dots \text{ Simple-condition-n.}$$

Examples of COMPOUND CONDITIONS are:

1. A = B and C > D
2. A < B or C = D
3. A < B or C < D and E = F.

## STATEMENTS

There are three types of statements: imperative statements, conditional statements, and compiler directing statements.

**Imperative Statements.** An imperative statement consists of a verb (excluding compiler directing verbs) and its operands, or a sequence of imperative statements. A sequence of imperative statements may contain either a GO imperative statement or a STOP RUN imperative statement, which (if present) must appear as the last imperative statement of its (GO or STOP RUN) sequence.

**Conditional Statements.** A conditional statement is defined to be one of the two following forms:

1. IF condition  $\left\{ \begin{array}{l} \text{statement-1} \\ \text{NEXT SENTENCE} \end{array} \right\}$   
 $\left\{ \begin{array}{l} \text{OTHERWISE} \\ \text{ELSE} \end{array} \right\} \left\{ \begin{array}{l} \text{statement-2} \\ \text{NEXT SENTENCE} \end{array} \right\}$   
 or
2. Imperative statement followed by a conditional statement.

In form 2 the imperative statement may not end with a GO or STOP RUN statement. When the IF SIZE ERROR option is used with the arithmetic verbs and when the IF END option is used with the READ verb, the verb, its operands, and the option is considered a conditional statement of the second form.

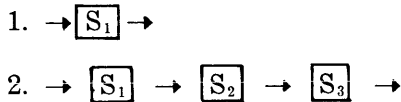
Statement-1 or statement-2 can be either imperative or conditional and, if conditional, can in turn contain conditional statements in arbitrary depth. If statement-1 or statement-2 is conditional, then the conditions within the conditional statement are considered "nested."

If statement-1 or statement-2 is missing from the conditional statement, the optional words NEXT SENTENCE may be substituted for the missing statement.

To illustrate, in COBOL the statement can be directly related to a path along a flow chart which has but one entry. The imperative statement has but one exit from the path. The conditional statement represents a branch along the path, and therefore represents a flow path that has more than one exit. To illustrate, if imperative statements are represented by S1, S2, S3,...Sn, and conditions by C1, C2, C3,...Cn, the following portions of flow charts represent imperative and conditional statements.

## IMPERATIVE STATEMENTS

FLOW CHART

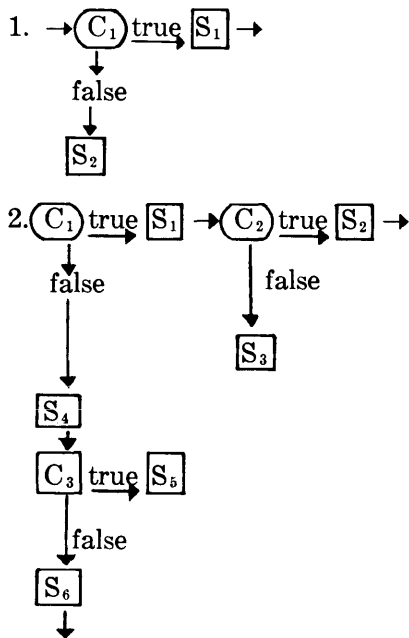


NARRATIVE

1. S<sub>1</sub>
2. S<sub>1</sub>;S<sub>2</sub>;S<sub>3</sub>

CONDITIONAL STATEMENTS

FLOW CHART



NARRATIVE

1. IF C<sub>1</sub>, S<sub>1</sub> OTHERWISE S<sub>2</sub>
2. IF C<sub>1</sub> S<sub>1</sub> IF C<sub>2</sub> S<sub>2</sub>  
ELSE S<sub>3</sub> ELSE S<sub>4</sub>  
IF C<sub>3</sub> S<sub>5</sub> ELSE S<sub>6</sub>

**Compiler Directing Statement**

A compiler directing statement consists of a compiler directing verb and its operands.

**SENTENCES**

A sentence consists of a sequence of one or more statements, the last of which is terminated by a period. The statements comprising the sentence must be either (a) one compiler directing statement or (b) an imperative or conditional statement syntactically correct according to the rules for statement formation.

A sentence which is composed of a compiler directing statement is called a compiler directing sentence. A sentence which is composed of an imperative or a conditional statement is called a procedural sentence.

**Imperative Sentences**

An imperative statement terminated by a period is an imperative sentence.

EXAMPLES:

- MOVE A TO B.
- MOVE A TO B; ADD C TO D.

An imperative sentence can contain either a GO statement or a STOP RUN statement, which (if present) must be the last statement in the sentence.

EXAMPLE:

- MOVE A TO B; ADD C TO D THEN GO TO START.

**Conditional Sentences**

A conditional statement terminated by a period is a conditional sentence.

EXAMPLES:

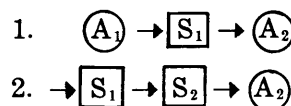
- IF X EQUALS Y THEN MOVE A TO B; OTHERWISE MOVE C TO D.
- IF X EQUALS Y MOVE A TO B IF W EQUALS T ADD A TO B; OTHERWISE NEXT SENTENCE; OTHERWISE MOVE C TO D.

If the phrase OTHERWISE NEXT SENTENCE immediately precedes the period, then the phrase OTHERWISE NEXT SENTENCE may be eliminated. This rule may then be applied again to the resulting sentence.

For example, in COBOL the sentence can be directly related to a portion of flow chart which has one and only one defined entry and one or more defined exits. No exit within a part of a flow chart that represents a sentence can enter within that portion of the flow chart. A sentence, therefore, becomes a statement with the entry point and all exit points defined. If connectors are represented by A1, A2, A3,..., the following flow charts represent imperative and conditional sentences.

**IMPERATIVE SENTENCES**

FLOW CHART

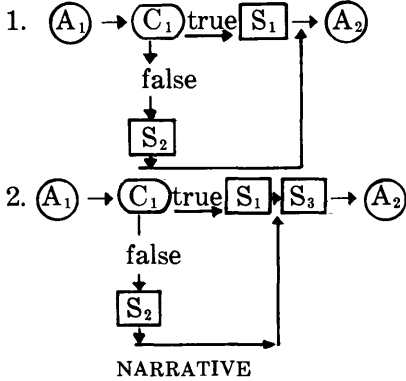


NARRATIVE

1. A<sub>1</sub>. S<sub>1</sub> GO TO A<sub>2</sub>.
2. S<sub>1</sub>. S<sub>2</sub> GO TO A<sub>2</sub>.

CONDITIONAL SENTENCES

FLOW CHART

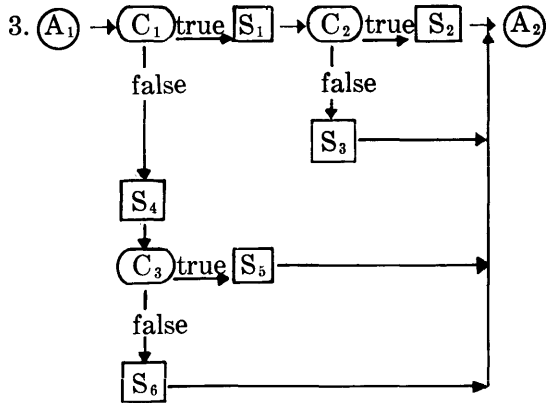


NARRATIVE

1. A<sub>1</sub>. IF C<sub>1</sub> S<sub>1</sub>  
OTHERWISE S<sub>2</sub>.
2. A<sub>1</sub>. IF C<sub>1</sub> S<sub>1</sub>  
ELSE S<sub>2</sub>. S<sub>3</sub> GO TO A<sub>2</sub>.

In this example two sentences are required to express the flow chart because the definition of a sentence does not permit internal connectors. In this example the entry connector to the sentence S<sub>3</sub>, GO TO A<sub>2</sub>, is unlabeled. The portion of a flow chart representing an unlabeled sentence can only be entered from that portion of the flow chart representing the preceding sentence.

FLOW CHART



NARRATIVE

3. A<sub>1</sub>. IF C<sub>1</sub> S<sub>1</sub>

IF O<sub>2</sub> S<sub>2</sub> ELSE S<sub>3</sub>

ELSE S<sub>4</sub> IF C<sub>3</sub> S<sub>5</sub>

ELSE S<sub>6</sub>.

The above definitions provide a method for converting flow charts to procedures. The rules can be more easily stated if, before translation occurs, all multiple connectors are named.

**RULES FOR CONVERTING FLOW CHARTS INTO NARRATIVE FORM**

1. List all named entry connectors in arbitrary order. Choose first entry.
2. Copy entry connector name, follow by period and erase first entry from entry connector list.
3. Follow flow line to next symbol.
4. If imperative box then copy statement; go to (3).
5. If condition box then copy IF condition number box and put number in condition box list; from true side of condition box go to (3).
6. If exit connector then copy GO TO exit connector name; go to (7).
7. If condition box list not empty then copy OTHERWISE; erase last entry from condition box list; from false side of condition box erased go to (3). If condition box list empty, copy period; go to (8).
8. If entry connector list not empty choose next entry, then go to (2). If entry connector list empty, stop.

There are rules for omitting some of the sentence names during translation to the narrative form.

**RULES FOR OMITTING NAMES**

1. If any reference (GO TO exit connector name) within a conditional sentence refers to the following sentence, then NEXT SENTENCE may replace "GO TO name" in that sentence.
2. In any conditional sentence a final OTHERWISE NEXT SENTENCE may be omitted.
3. A sentence name may be omitted if all references to that sentence are from the preceding sentence.
4. The application of Rules 1, 2, and 3 require that the adjacency of the affected sentence be preserved.

**Punctuation**

A separator is a word or character used as a definition or for the purpose of enhancing readability. With the exception of the period, the use of separators is optional.

The allowable separators are



- . period
- , comma
- ; semicolon

THEN

Separators must be followed by a space and cannot be immediately followed by another separator.

Separators must be used in the following places:

- a. To terminate a sentence.
- b. To delimit a noun from the beginning of the entry it names for referencing purposes.

Separators may be used in the following places:

- a. Between procedural statements.
- b. In a conditional statement between:
  - (1) The "condition" and the statement.
  - (2) The statement and OTHERWISE
- c. As defined in the Verb Formats.

## SENTENCE EXECUTION

"Execution of a sentence or a statement within a sentence" means "execution of an object program compiled by the B 5000 Programming System from a sentence, or from a statement within a sentence which has been written in COBOL." "Transfer of control" means "transfer of control in the object program by transferring GOing) from one place (control point) to another place (control point) out of the written sequence." "Passing of control" means "passing of control in the object program by passing from one place (control point) to the next place (control point) in the written sequence."

### Imperative Sentence

An imperative sentence is executed in its entirety and control is passed to the next sentence.

Whenever a GO statement is encountered during execution of a sentence or statement, there will be an unconditional transfer of control to the sentence referenced by the GO statement.

### Conditional Sentence

```

IF condition {statement-1
                { [NEXT SENTENCE] }
}
{ ELSE
  { OTHERWISE } } {statement-2.
                    { [NEXT SENTENCE] }
}

```

In the conditional sentence above, the "condition" is an expression which has the value true or false. If the condition is true then statement-1 is executed and control is transferred to the next sentence. If the condition is false, statement-2 is executed and then control is transferred to the next sentence. If state-

ment-1 is imperative and contains a GO statement, then, if the condition is true, statement-1 is executed and control is transferred to the sentence referenced by the GO statement. Similarly if statement-2 is imperative and contains a GO statement, then, if the condition is false, statement-2 is executed and control is transferred to the sentence referenced by the GO statement.

### Compiler Directing Sentences

A compiler directing statement terminated with a period is a compiler directing sentence.

EXAMPLE:

```

USE procedure-1 THROUGH procedure-2
AFTER STANDARD ERROR PROCEDURE
ON INPUT.

```

Compiler directing sentences direct the B 5000 Programming System to take action at compilation time (for example, INCLUDE A PROCEDURE from the LIBRARY or USE named PROCEDURES AFTER STANDARD ERROR PROCEDURES). On the other hand, procedural sentences denote action to be taken in accomplishment (executors) of the procedures by the object program at object time.

Compiler directing sentences may direct inclusion of generated programs in the object program. However, compiler directing sentences do not directly result in either transfer or passing of control. However, the generated program itself, which compiler directing sentences may have included in the object program, is subject to the same rules for transfer or passing of control as if that generated program had been created from procedural sentences only.

## CONTROL RELATIONSHIP BETWEEN SENTENCES

In COBOL, imperative and conditional sentences describe the procedure that is to be accomplished. The sentences are written successively, according to the rules of the Reference Format, to establish the sequence in which the object program is to accomplish (execute) the procedure. In order to vary the sequence in which the object program is to execute the sentences which comprise a procedure, a sentence may be named. The name consists of a noun followed by a period. The name precedes the sentence it names. Hence, in the Procedure Division, names are used so that one procedure can refer to (reference) another procedure by naming (referencing) it.

In executing sentences, control is transferred only to the beginning of a sentence. Control is transferred to an unnamed sentence only from the sentence written immediately preceding the unnamed sen-

tence. If a sentence is named, control is transferred to it from either (a) the sentence immediately preceding the unnamed sentence, or (b) from any sentence which contains a GO TO followed by the name of the sentence to which control is to be transferred.

## PARAGRAPH

So that the source programmer may group several sentences to convey one idea (procedure), paragraphs have been included in COBOL. In writing procedures in accordance with the rules of the Procedure Division and the requirements of the Reference Format, the source programmer begins a paragraph with a named sentence. Hence, the sentence name can be considered and becomes the paragraph name.

A paragraph consists of one or more successive sentences, the first and only the first of which is named. A named sentence, therefore, cannot appear within a paragraph as other than the first sentence of that paragraph.

The source programmer will place compiler directing sentences in their own paragraphs. Paragraphs comprised of compiler directing sentences are called "compiler directing paragraphs."

Paragraphs which contain at least one procedural sentence are called "procedural paragraphs."

## SECTION

A section consists of one or more paragraphs and when designated must be named. The section name is followed by the word SECTION, a priority number which is optional, and a period. The section name applies to all paragraphs following it until another section name is found.

## ACKNOWLEDGMENT

This publication is based on the COBOL Specifications developed by a voluntary committee composed of government users and computer manufacturers. The following organizations have participated in the development of the COBOL Specifications:

Bendix Corporation, Computer Division

Burroughs Corporation  
Computer Science Corporation  
Control Data Corporation  
Department of Commerce, Bureau of Standards  
General Electric Company, Computer Division  
International Business Machines Corporation  
Minneapolis-Honeywell Corporation,  
Data Processing Division  
National Cash Register Company,  
Computer Division  
Philco Corporation  
Radio Corporation of America  
Remington Rand Univac, a Division of  
Sperry Rand Corporation  
Sylvania Electric Products, Incorporated  
U. S. Air Force, Air Materiel Command  
U. S. Navy, David Taylor Model Basin,  
Bureau of Ships

Ideas and information were drawn from many sources: in particular, from the FLOW-MATIC\* System developed by Sperry Rand, the Commercial Translator System designed by IBM, the AIMACO System developed jointly by the Air Materiel Command and Sperry Rand, and the FACT\*\* System developed by Computer Science Corporation for the Minneapolis-Honeywell Corporation. With the permission of the authors and publishers, certain material has been taken from the following copyrighted publications: FLOW-MATIC\* Programming System, © 1958 Sperry Rand Corporation, and General Information Manual: IBM (Commercial Translator),\*\* © 1959 by International Business Machines Corporation: FACT\*\*\* System, © 1959 by Minneapolis-Honeywell Corporation.

The initial specifications for the COBOL language were the result of contributions made by all of the above mentioned organizations and no warranty expressed or implied as to the accuracy and functioning of the programming system and language is made by any contributor or by the committee in connection therewith.

\*Trademark of Sperry Rand Corporation

\*\*Trademark of IBM Corporation

\*\*\*Trademark of Minneapolis-Honeywell Corporation



# APPENDIX C

## GLOSSARY

(Some terms defined are characteristic to the industry, but others are significant only with respect to the B 5000 System. For a more complete glossary of industry terminology it is recommended that reference be made to glossaries published in such trade journals as ACM Communications and Computers and Automation.)

**Address**—A label, such as an integer or other set of characters, which identifies a memory location or storage device.

**ALGOL**—(for ALGO<sup>r</sup>ithmic Language) an international problem language designed for the concise, efficient expression of arithmetic and logical processes, and the control (iterative, etc.) of these processes.

**Algorithm**—A statement of the steps to be followed in the solution of a problem.

**Argument**—Known reference factor necessary to find the desired item in a table or array. Sometimes referred to as a “key” as in “search key.”

**Array**—An ordered arrangement of items of information.

**Automatic Programming**—Technique which employs the computer itself to translate programming from a form that is easy for a human being to produce and understand into a form suitable for use by a computer.

**Binary**—A radix-2 number system using only the digits 0 and 1.

**Boolean Algebra**—A system of algebra dealing with truth values as variables and having basic operators such as “and,” “or,” “not,” etc.

**Boolean Variables**—An operand in a Boolean algebra expression. A Boolean variable may have the value of “true” or “false,” commonly represented in computers by one and zero respectively.

**BURROUGHS Common Language (BCL)**—A binary code representation of alphanumeric characters common to all future BURROUGHS equipment and common with existing standard punched-card and tape representations.

**Call**—A set of characters or bits which demand an action to take place or some item of information; for example, subroutine call, operand call, descriptor call.

**Channel**—A path along which information may flow. (See Input/Output Channel.)

**Characteristic**—The exponent portion of a floating-point number. (See Floating-Point Representation.)

**Clock**—A time-increment counting register used for program-interrupt and job-time accounting.

**COBOL**—A CO<sup>m</sup>mon BU<sup>s</sup>iness OR<sup>i</sup>ented Language designed for expressing problems of data manipulation and processing in English narrative form.

**Compiler**—A translator program which reduces a problem-oriented language into the machine language of a particular computer.

**Concatenating**—Linking together by forming a chain or series, a series or order of things depending on each other.

**Control Counter**—A 17-bit register which indicates the location of the next syllable to be executed by the Processor.

**Data Array**—Any ordered set of data, such as the information on a card, a tape record, a print line, the contents of a working area, etc.

**Data-Manipulation Mode**—One of the logical operational modes of the B 5000, in which the basic information unit is a single alphanumeric character. Utilized for efficient editing, formatting, and comparison functions.

**Data Manipulators**—A set of operators which edit, compare, and move data within memory when the Processor is in the Data-Manipulation mode.

**Debug**—To isolate and correct the mistakes in a program.

**Descriptor**—A computer word used specifically to define characteristics of a program element. For example, descriptors are used for describing a data record, a segment of a program, or an input-output operation.

**Descriptor Call Syllable**—A syllable of the B 5000 program string which directs the Processor to place in the Stack the location of a data array or a program segment.

**Diagnostic Routine**—Routine designed to detect and locate either a malfunction of the system or a mistake in programming.

**Drum, Magnetic**—A rapidly rotating cylinder, the

- surface of which is coated with a magnetic material on which information may be stored as small magnetized areas.
- Edit**—The act of arranging information from input-output devices. This may involve the selection of pertinent data, the insertion of symbols such as page numbers and check-protection characters, and standard processes such as zero suppression.
- Executive Routine**—A routine designed to control and cause the execution of other routines. (See Master Control Program.)
- Field**—A set of one or more characters which is treated as a unit of information.
- Fixed-Point Representation**—An arithmetic notation in which all numeric quantities are expressed by the same number of digits with the decimal point (for base 10) or octal point (for base 8) assumed in a fixed location in each number. Alignment of numbers with different assumed locations of the points must be performed by the program before an arithmetic operation such as addition can be performed.
- Floating-Point Representation**—An arithmetic notation in which all numeric quantities have an associated indication of the decimal point location (base 10) or octal point location (base 8). Automatic alignment of numbers and calculation of the location of the point can be provided in arithmetic on floating-point numbers. In the B 5000, a floating-point number consists of two parts: a 13-digit octal integer with sign called the mantissa; and a signed number called the characteristic (or exponent) which indicates the number of places to the right or left that the actual octal point is from the assumed octal point in the mantissa.
- Hardware**—The mechanical, magnetic, electrical, and electronic devices from which a computer system is constructed.
- Housekeeping**—Operations not directly concerned with the objective of a program; e.g., packing or rearranging data, subroutine linkages, etc.
- Indirect Address**—An address which identifies a memory cell containing an address. The contents of the memory cell is the address of the desired information or may also be an indirect address.
- Input/Output Channel**—A device which allows independent, simultaneous communication between any Memory Module and any of the several input-output units. It controls any peripheral device and performs all validity checking on information transfers.
- Input/Output Exchange**—An electronic switch which connects an Input/Output Channel to the designated peripheral device.
- Interrupt**—A signal generated by an input-output device, by an operational error, or by a request by the Processor for more data or program segments. Provides the Master Control Program with the facility to maintain control of all system functions.
- Jump**—An operation which may alter the normal sequence of a program. Normally syllables are executed in sequence; a jump operation causes a termination of the sequence and directs the Processor to a specified syllable. A conditional jump operation is a jump operation which takes place only if a specific condition exists in the Processor. Usually the condition is a result of a test or comparison operation. If the specific condition does not exist, a conditional jump operator is ignored and sequential execution of syllables continues.
- Library**—Collection of fully tested standard programs and subroutines for repeated use by, or incorporation into, other programs.
- Literal**—An element in a program which is itself a quantity or alphanumeric constant to be used by the program rather than being an address of the quantity or constant.
- Machine Language**—The coded operations that control information and addresses employed within the Processor to express a program. (See Problem Language.)
- Mantissa**—Integer part of a floating-point number (13 octal digits in a single-precision number of the B 5000). (See Floating-Point Representation.)
- Master Control Program**—A computer program to control the operation of the system. It is designed to reduce the amount of intervention required of the human operator. The Master Control Program performs the following functions: schedules programs to be processed; initiates segments of programs; controls all input-output operations to insure efficient utilization of each system component; allocates memory dynamically; issues instructions to the human operator and verifies that his actions were correct; performs corrective action on errors in a program or system malfunction.
- Megacycle/Sec.**—A million cycles per second. The basic pulse rate of the B 5000 is 1 megacycle/second.
- Memory**—Internal computer storage. Distinguished from other types of storage in the B 5000 which are part of the peripheral equipment.
- Memory Exchange**—An electronic switching device which controls information flow among Memory

- Modules and the Processor or Input/Output Channels.
- Microsecond—One millionth of a second (0.000001 sec. or 1  $\mu$ s).
- Modularity—The property of a system resulting from the construction or assembly of the system from logical subunits (modules). In the B 5000, this property provides the capability of constructing a system with the proper number of each type of module to match varying processing requirements efficiently and to maximize the utilization of each module.
- Module—A logical subunit that may be easily detached from, or included with, the whole system. Processor, Magnetic Tape Units, and Storage Drums are typical modules of the B 5000 System.
- Multi-Processing—Processing several programs or program segments concurrently on a “time-share” basis. The Processor is only active on one program at any one time while operations such as input-output may be performed in parallel on several programs. The Processor is directed to switch back and forth among programs under the control of the Master Control Program.
- Nesting—Enclosing one program element of a particular type, such as a subroutine, within another of the same type.
- Noisy Mode—A mode of floating-type arithmetic operation in which the error resulting from use of only a finite number of significant digits may be identified.
- Normal Mode—The standard B 5000 operational logic used during computational processes. The basic information unit is the word.
- Object Program—A set of machine-language instructions for the solution of a specified problem, obtained as the end result of the compilation process (see Compiler, Problem Language).
- Octal—A number system based on powers of 8 rather than 10 as in the decimal system. Includes only the digits 0, 1, 2, 3, 4, 5, 6, and 7.
- Operand—Any of the quantities entering into an operation. An operand is typically a number for arithmetic operations. For comparison operations, an operand may be an alphanumeric field.
- Operand-Call Syllable—A syllable which specifies that an operand be brought to the Stack, either directly from the Program Reference Table or indirectly by means of a descriptor.
- Operators—Symbols that denote a fixed, predefined set of operations to be performed in a specified sequence. There are a number of classes of operators in the B 5000: for example, the arithmetic operators are +, −, ×, /, DIV; the relational operators are <, ≤, =, >, ≥, ≠.
- Output Channel—(See Input/Output Channel.)
- Parallel Operation—Flow of data through the system or any part of it, using two or more communication lines or channels simultaneously.
- Parallel Plate Packages—A packaging technique for logical (and other) computer circuitry developed by BURROUGHS to achieve high packing density, ease of automatic production and assembly, and simple maintenance.
- Parallel Processing—Processing more than one program at a time on a parallel basis, where more than one Processor is active at a time (distinguished from Multi-Processing where only one Processor is active on one program at a time).
- Parameter—In a subroutine, a quantity which may be given different values when the subroutine is used in different parts of one main routine but which usually remains unchanged throughout any one such use. To use a subroutine successfully in many different programs requires that the subroutine be adaptable by changing its parameters.
- Parity Check—A summation check in which the binary digits, in a character or word, are added (modulo 2) and the sum checked against a single, previously computed parity digit.
- Peripheral Equipment—Any of the several devices, primarily used to communicate with a system, not considered a part of the main processing and control system. On the B 5000, the peripheral equipment includes Magnetic Tape Units, Line Printers, Card Readers, Card Punches, Keyboard, Message Printer, and Plotter.
- Polish Notation—A method of writing logical and arithmetic expressions without the need for parentheses, originated by the Polish logician J. Lukasiewicz. For example: Normal algebraic notation  $(X + Y) \times (A - B)$ , in Polish notation:  $XY + AB - \times$ .
- Precision—The degree of exactness with which a quantity is stated. For example, the number 2.783 is precise to four digits, but does not necessarily have four digits of accuracy.
- Presence Bit—A single flag bit appearing in descriptors to indicate whether or not the information to which reference is made by the descriptor is in high-speed (core) memory at this time.
- Priority—A value assigned to a program or program segment to specify the relative processing se-

- quence. The priorities of all programs to be run are taken into consideration by the Master Control Program in arriving at a schedule.
- Program Independent Modularity**—Property of the B 5000 to accept changes in system configuration and adjust programs accordingly to yield maximum utilization of all modules without reprogramming or recompilation of programs.
- Problem Language**—The language used by the programmer to state the definition of a problem. ALGOL and COBOL are examples of problem languages. Problem languages are closely related to the type of problem being stated—i.e., algebraic statements for mathematical problems (ALGOL) and narrative English statements for commercial problems (COBOL). Problem language should not be confused with machine language. A program is written in problem language by the programmer. This source program is then translated to the object program (in machine language) by a compiler program. (See Object Program, Compiler.)
- Program (noun)**—A plan for the solution of a problem. A B 5000 program may be a statement of the problem in ALGOL or COBOL or the translated, segmented object (compilation result) program.
- Program (verb)**—To plan a computation or process from the original statement of the problem to the delivery of the results, including the integration of the operation of the resulting program into an existing system (for conventional computers). For the B 5000: A system analysis and statement of the problem in common language.
- Programming System**—The B 5000 Programming System consists of the ALGOL and COBOL compiler and the Master Control Program. The ALGOL and COBOL compiler is loaded by the Master Control Program.
- Program Reference Table (PRT)**—An area in memory for the storage of operands, references to operands, references to segments of a program, and other program variables. Permits programs to be independent of the actual memory locations occupied by data and parts of the program. Thus programs and data can be placed into any available memory areas without modification to the program.
- Real Variable**—A variable over the rational and irrational classes of numbers. In ALGOL a real variable is a floating-point number as distinct from an integer variable which is an integer.
- Register**—The hardware for storing one or more computer words or for maintaining internal system control.
- Relocatability**—A facility whereby programs or data may be located any place in memory at different times without requiring modification to the program. In the B 5000, segments of the program and all data are independently relocatable with no loss in efficiency.
- Return**—An operator in a subroutine which recovers all pertinent information from the Stack and transfers control to the next syllable in the original syllable string of the program which causes entry to the subroutine.
- Return Point**—The syllable in the program segment to which control is transferred after the completion of a subroutine or an intercession by the Master Control Program.
- Scheduling**—Designation of times and sequence of projected operations. One of the functions of the B 5000 Master Control Program.
- Segment (verb)**—To divide a program into an integral number of parts, each of which performs some part of the total program and is capable of being completely stored in internal memory.
- Simultaneity**—Concurrent communication between various units of a system at the same instant.
- Software**—Programs, routines, and procedures which augment and support a computer system (the Master Control Program, compilers, etc.).
- Stack**—A portion of memory and/or registers used for temporarily holding information. A Stack, as used in the B 5000, operates on the “last-in first-out” principle, that is, the last item of information placed in the Stack will be the first item of information used when information is required from the Stack. Operators perform their operations on information at the top of the Stack. (See Operators.)
- Storage**—Any device into which information can be copied, which holds this information, and from which the information can be obtained at a later time.
- Storage Allocation**—Assignment of specific memory addresses to individual program elements (done automatically in the B 5000 at object running time by the Master Control Program).
- Subroutine**—The set of instructions necessary to carry out a defined operation; a subunit of a program.
- Subroutine Call**—A set of characters or lists which initiate a subroutine and contain the parameters or identification of the parameters required by



# Burroughs Corporation

DETROIT 32, MICHIGAN

IN CANADA, BURROUGHS ADDING MACHINE OF CANADA, LTD., TORONTO, ONTARIO