A NARRATIVE DESCRIPTION
of the

Burroughs
B 5500

DISK FILE
MASTER CONTROL PROGRAM

REVISED
OCTOBER, 1966

A NARRATIVE DESCRIPTION

of the

Burroughs B 5500

DISK FILE MASTER CONTROL PROGRAM

Business Machines Group

Burroughs Corporation

Systems Documentation

Burroughs Corporation
Detroit, Michigan 48232

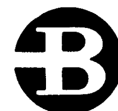


TABLE OF CONTENTS

SECTION	TITLE	PAGE
	INTRODUCTION	ix
1	BURROUGHS B 5500 INFORMATION PROCESSING SYSTEM	1
	General	1
	Stack	1
	Program Reference Table (PRT) and Relative Addressing	2
	Relative Address	2
	Operand Call	3
	Store Operators	3
	Notation Used for Word Description	5
	Data Words	5
	Operands	5
	Data Descriptors	5
	Bit Explanation	6
	Identification Bit	6
	Presence Bit	6
	Size Field	6
	Address Field	6
	Operand Call and Data Words	6
	Operand Call on an Operand	6
	Operand Call on a Data Descriptor	6
	Indexing	7
	Polish Notation	7
	Examples of Polish Notation	9
	Syllables: Instructions for the B 5500	10

TABLE OF CONTENTS (Cont'd.)

SECTION	TITLE	PAGE
1	Operand Call Syllable (10)	10
	Descriptor Call Syllable (11)	10
	Literal Call Syllable (00)	10
	Operator Syllable (01)	10
	Polish Notation and B 5500 Programs	10
	Programs for the B 5500: Segmented and Address Independent	11
	C Register: Address of Control	11
	Examples of Statements and Resulting Code	12
	Transferring Control within Segments: Relative Branching	13
	Transferring Control to Other Segments and the Operand Call	13
	Procedures: Subroutines of the B 5500	14
	F Register	14
	Control Words for Procedures	15
	Mark Stack Control Word	15
	Return Control Word	15
	Procedure Descriptor	15
	Calling a Procedure	15
	Mark Stack Operator	15
	Operand Call on a Procedure Descriptor	15
	Executing Procedure	16
	Relative Addressing in the Sub-Program Level	16
	Exiting a Procedure	17
	Additional Remarks about Subroutines	18
	B 5500 Processors and States	18
	Normal State	18
	Control State	18

TABLE OF CONTENTS (Cont'd.)

SECTION	TITLE	PAGE
1	Interrupts	18
	Processor Dependent and Processor Independent Interrupts	19
	Action to Handle Interrupts	19
	Interrupt Control Word	19
	Interrupt Return Control Word	19
	Initiate Control Word	19
	Interrogate Interrupt Operator	20
	I/O on the B 5500	20
	Initiating an I/O	20
	I/O Descriptor	21
	Operation of an I/O Control Unit	21
	I/O Completion	21
	I/O Result Descriptor	21
	Handling an I/O Finish Condition	21
	A Note on Multiprocessing	21
2	THE DISK	23
	General	23
	User Disk	23
	Program Files vs. Data Files	23
	Format of Files on Disk	23
	System Disk	23
	Requirements	23
	Overlay Storage	23
	Disk Directory	23
	Available-Disk Table	24
	Initializing the Disk	24

TABLE OF CONTENTS (Cont'd.)

SECTION	TITLE	PAGE
3	DISK FILE MASTER CONTROL PROGRAM	25
	General	25
	Initiating the B 5500 Disk File System	25
	DF MCP Classification and Organization of Core Storage	25
	Non-Overlayable Storage	25
	Overlayable Storage	26
	Overlayable Program Segment Areas	26
	Overlayable Data Areas	26
	Available Storage	26
	Memory Links	26
	Memory Links for In-Use Storage	26
	Memory Links for Available Storage	27
	Creating the Available-Disk Table	27
	Interrogating Peripheral Units	27
	STATUS Procedure	27
	CONTROL CARD Procedure	28
	SELECTION Procedure	28
	RUN Procedure	28
	MIX Index	28
	INITIATE Routine	29
	P1MIX and P2MIX	29
	Information Source for a Program In Process	29
	PRESENCE BIT Routine	29

TABLE OF CONTENTS (Cont'd.)

SECTION	TITLE	PAGE
3	Methods of Making Information Present	30
	Making Data Present for the First Time	30
	Making Overlaid Data Present	30
	Making Program Segments Present	30
	Control Section	31
	INDEPENDENT RUNNER Procedure and SLATE Array	31
	SLEEP Procedure and BED Array	31
	NOTHINGTODO Routine	32
	GETSPACE Procedure	32
	Locating an Area for Overlayable Storage	33
	Locating an Area for Non-Overlayable Storage	33
	OLAY Procedure	33
	Overlaying a Data Area	33
	Overlaying Program Segment Areas	34
	Overlaying a DF MCP Segment	34
	FORGETSPACE Procedure	35
	ESPBIT Procedure	35
	Object Program I/O Facilities	36
	I/O Intrinsics	36
	Specification of File Handling Techniques	36
	File and File Names	36
	File Parameter Block	36
	File Information Blocks	37
	Logical Unit Numbers	37
	File Names vs. I/O Units	37

TABLE OF CONTENTS (Cont'd.)

SECTION	TITLE	PAGE
	Opening a File	38
	Buffer Area Accessed by Object Programs	38
	Communicate	38
	Performance of Object Program I/O by the DF MCP	39
	PROGRAM RELEASE Procedure	39
	CONTINUITY BIT Routine and PRINTER BACKUP Procedure	39
	IOREQUEST Procedure	39
	INITIATEIO Procedure	40
	IOFINISH Procedure	40
	A Note on Parallel Processing	40
	Breakout, Restart, and Emergency Interrupt Facilities	41
	Breakout and Emergency Interrupt	41
	REDUMP Procedure	42
	Restart	42
	Characteristics of the Program Logging Facility	43

LIST OF ILLUSTRATIONS

FIGURE	TITLE	PAGE
1	Stack Movement	2
2	Relative Addressing	3
3	Destructive Store Operation	4
4	Bit Reference	5
5	Operand Call Accessing a Data Descriptor	8
6	Sub-Program Level Relative Addressing	17

INTRODUCTION

The B 5500 Electronic Information Processing System is the result of a hardware-software integration. Processing on a B 5500 System results from the interrelated actions of: (1) object programs produced by B 5500 compilers for problem-oriented languages, (2) a master control program, and (3) the B 5500 hardware.

The primary purpose of this document is to provide a description of the operational characteristics of the Disk File Master Control Program (DF MCP) for Burroughs B 5500 Disk File System. However, because of the B 5500 hardware-software interrelationship, a description of the DF MCP can proceed only under the assumption that the reader is at least basically familiar with the operational characteristics of the B 5500. Also, the reader should be familiar with the organization of disk storage. Consequently, this publication is divided in three sections; Section 1 is a discussion of operational characteristics of the B 5500, Section 2 is a discussion of the organization and use of disk storage, and Section 3 is a discussion of the operational characteristics of the DF MCP. It may be helpful, but not necessary, for the reader to be familiar with the problem oriented languages of the B 5500.

SECTION 1

BURROUGHS B 5500 INFORMATION PROCESSING SYSTEM

GENERAL

The following discussion covers operational characteristics of the Burroughs B 5500 Information Processing System. It is not a complete description of all characteristics; however, it should provide the information required to understand the operations of the Disk File Master Control Program (DF MCP). Only word mode operation is discussed.

STACK

Every program in process on the B 5500 has its own stack. The stack is used for various purposes during the execution of a program and it is important to be familiar with its characteristics.

The stack of a program currently executing consists of: (1) the arithmetic registers rA and rB and (2) a reserved area of core memory immediately preceding the program's Program Reference Table which will be discussed later. The association between the core memory portion of the stack and registers rA and rB is established by the S register. The stack has the following characteristics (r means register):

1. The top of the stack is considered to be the stack location containing the most currently obtained value.
2. Information being placed in the stack is placed in rA.
3. If information is to be placed in the stack (i.e., placed in rA) and rA is full, the information in rA is first automatically placed in rB.
4. If information from rA is to be placed in rB and rB is full, the information in rB is first automatically placed in the next available location in the core memory portion of the stack.
5. The address of the word most recently placed in the core memory portion of the stack is addressed by rS.
6. When a value from rB is to be placed in the core memory portion of the stack, the value of rS is first incremented by 1.
7. When information in the stack is to be used, it is used on a last-in, first-used basis (i.e., from the top of the stack).
8. Various operations cause the information in rA and/or rB to be used, consequently leaving them empty; the processor has a means of then marking them "empty". If information is required in rA and/or rB, and the information is not where it is required, then the flow of information is the reverse of that described above until the desired conditions are met. For example, if rA was empty and rB full, and a particular operation required that both rA and rB be full, then the following actions would occur to meet the conditions for the operation (figure 1):

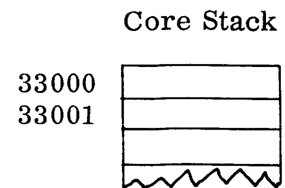
   a. The information in rB would be placed in rA.

   b. The word addressed by rS would be placed in rB.

   c. Register rS would be decremented by 1.

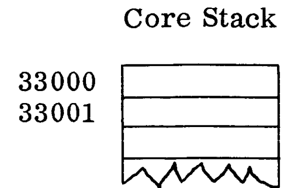
Step 1: Place P1 in Stack

rS 33000 | rA P1 | rB Empty



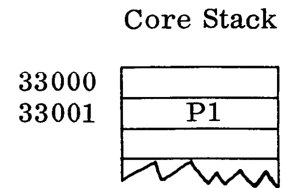
Step 2: Place P2 in Stack

rS 33000 | rA P2 | rB P1



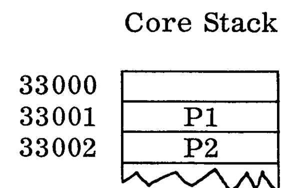
Step 3: Place P3 in Stack

rS 33001 | rA P3 | rB P2



Step 4: Place P4 in Stack

rS 33002 | rA P4 | rB P3



Step 5: Place P5 in Stack

rS 33003 | rA P5 | rB P4

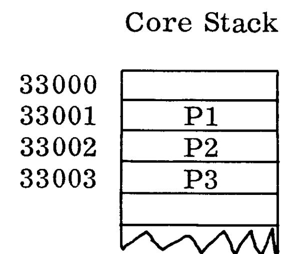


Figure 1. Stack Movement

Figure 1 shows the status of the stack and rS as values represented by P1, P2, P3, P4, and P5 are placed in the stack. Registers rA and rB are assumed empty at the start.

PROGRAM REFERENCE TABLE (PRT) AND RELATIVE ADDRESSING

Every program for the B 5500 has a PRT (Program Reference Table). The PRT contains the locations reserved for program variables, data descriptors which give information about data arrays, and other program information.

Relative Address

When a program references a word in its PRT, the relative address of the word is used, never the absolute address. The relative address of any particular location is based on its position relative to the beginning of the PRT. The first PRT word is word zero. This method of addressing is used because it does not rely on actual addresses that exist at run time. At run time, a relative address is related to an absolute address through use of rR.

When a program is executing, rR is set to the absolute address of the base of that

program's PRT, according to wherever it was read into core. Consequently, each relative address used to reference the PRT can be added to the value of rR to obtain the desired absolute address. This action is carried out automatically by the processor at run time when a location is referenced.

Operand Call

A principal method of obtaining values from a PRT, using the relative addressing technique, is through use of the operand call syllable. A B 5500 instruction is referred to as a syllable, and one B 5500 word accommodates four syllables. Each operand call syllable contains the relative address of the location from which it is to obtain information. When an operand call syllable is executed, this relative address is automatically added to the value in rR. The result, which is the absolute memory address of the pertinent location, is placed in the M register. The information addressed by rM is then obtained from the PRT and placed in the top of the stack (figure 2).

Figure 2 shows register and core conditions before and after an operand call addressing relative address 30 of the PRT which contains a value represented by X6.

Store Operators

Storing information in the PRT is also done through the use of relative addresses. Storing

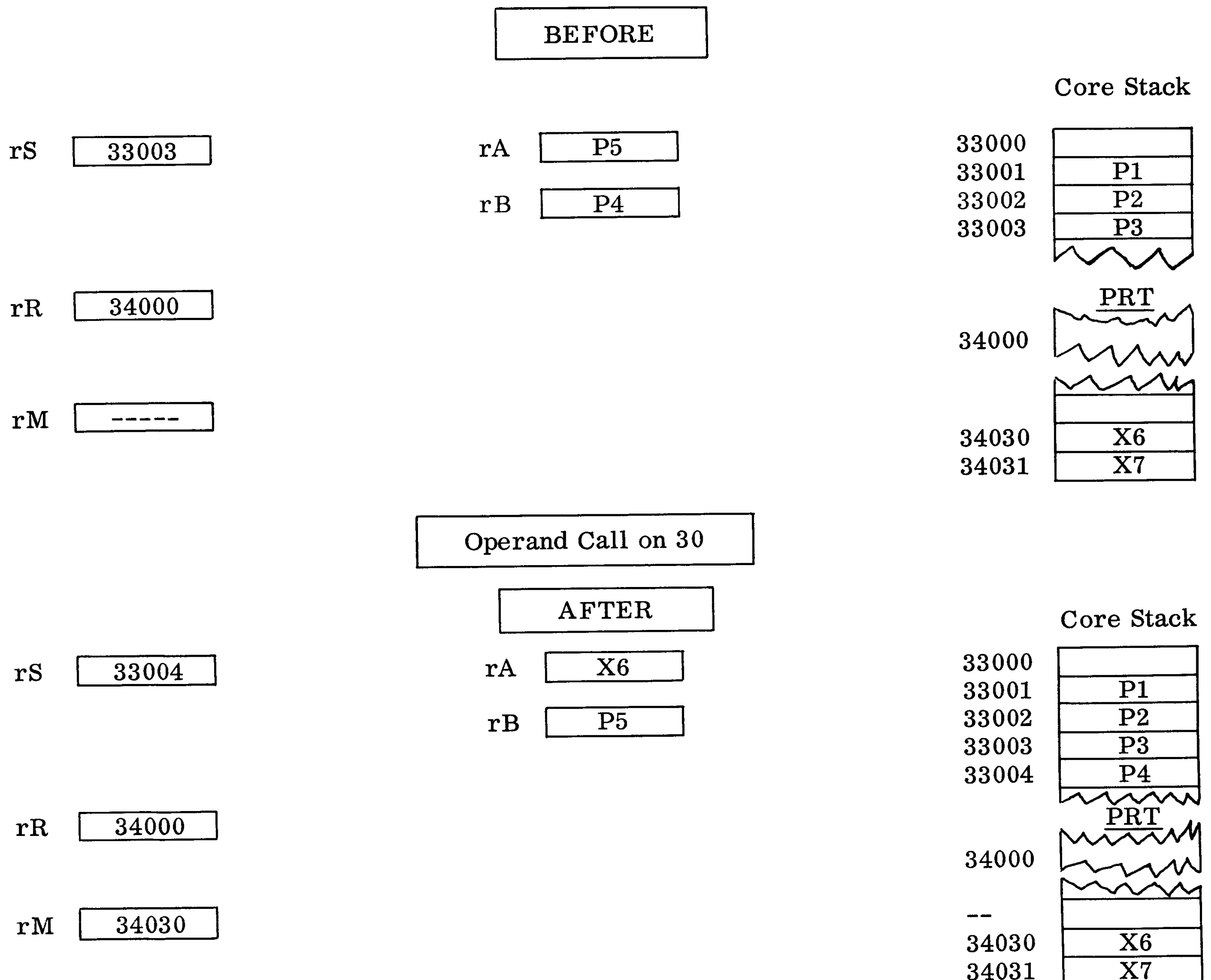


Figure 2. Relative Addressing

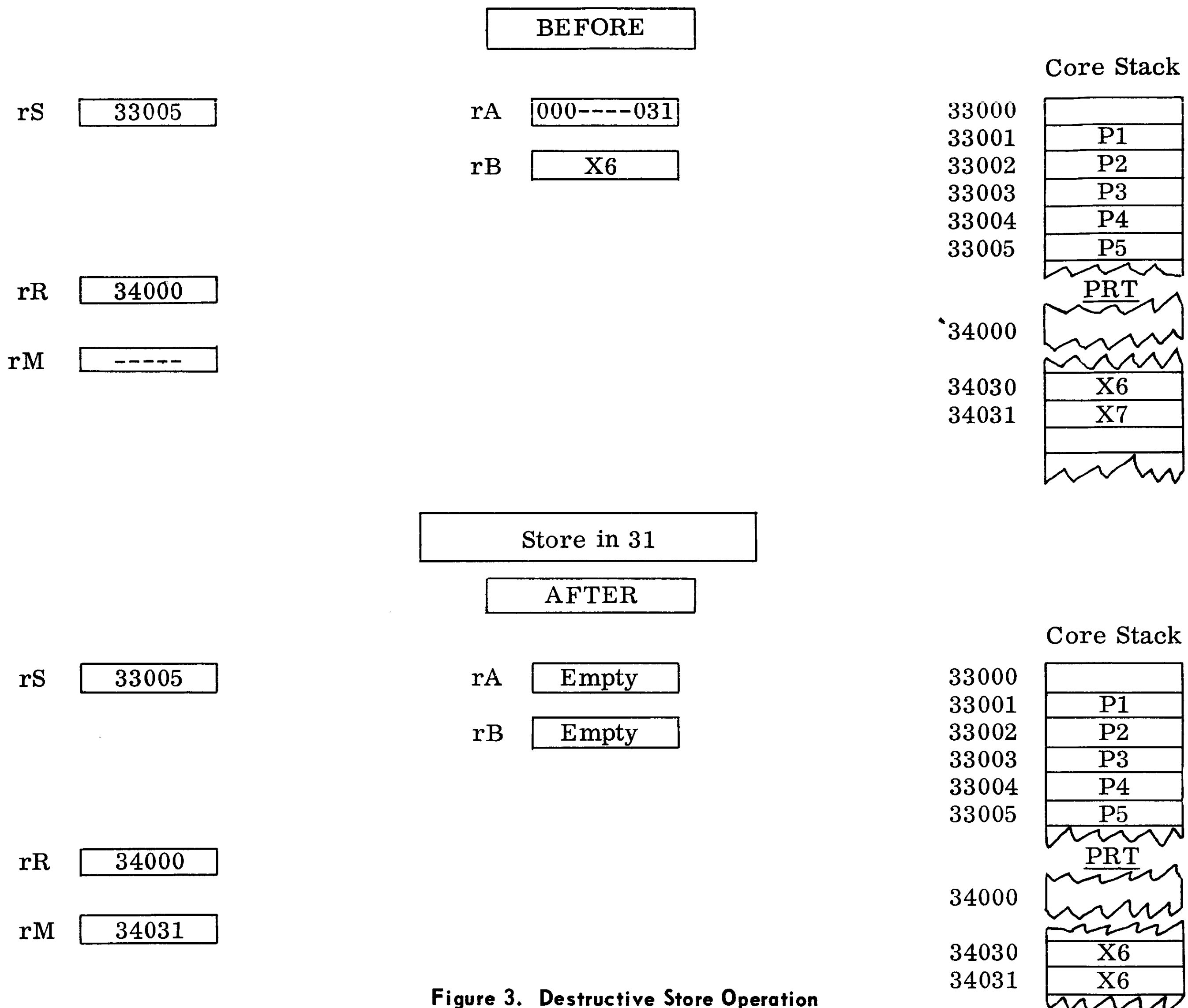


Figure 3. Destructive Store Operation

operations are carried out in the following manner.

To store a value in the PRT, a literal (i.e., an integer with a value from 0 to 1023) equal to the relative address of the pertinent PRT location must be the top word in the stack. The information to be stored must be the second from top word in the stack. With these conditions existing, a store operator can be executed and the following actions will occur automatically. The literal value (i.e., relative address) at the top of the stack is added to the value of rR, yielding the desired absolute address which is placed in rM. The value which was the second word in the stack is then stored in the location (in the PRT) addresses by rM (figure 3).

It should again be pointed out that the actual location of the PRT from run-time to run-time may change without affecting the program. This logically follows from the facts that: (1) a program only uses relative addresses to address the PRT, (2) the actual addresses of PRT locations are determined through use of rR, and (3) the DF MCP always sets rR to the base of the program's PRT, wherever it is in memory.

Figure 3 shows register and core conditions before and after a destructive store operation (i.e., a store operation that removes the value from the stack after storing it) referencing relative address 31 in the PRT. The value 31 has already been placed in the stack as a relative address; the value X6 is to be stored.

NOTATION USED FOR WORD DESCRIPTION

A B 5500 word is 48 bits in length. Reference to particular bits, made in this document, will be specified using the following conventions:

1. The bits in a word are referenced from left to right and are numbered from 0 (zero) through 47 (figure 4).

2. Reference to a particular bit or group of contiguous bits will be made using the construct:

   [INTEGER:INTEGER]

   where the INTEGER on the left of the colon specifies the left-most bit of the field and the INTEGER on the right of the colon specifies the number of bits in the field.

Figure 4 is a representation of a B 5500 word followed by a description of its contents.

DATA WORDS

The B 5500 recognizes two types of data words: operands and data descriptors. In appearance, the only difference between an operand and a data descriptor is in the value of one particular bit, the flag bit. The flag bit of a word is bit [0:1]. Operands have a flag bit of zero; descriptors have a flag bit equal to 1. In use, operands and data descriptors have different functions.

Operands

Basically, an operand is concerned with one memory location, the one which it occupies. An operand is recognized as a value, such as a floating point arithmetic quantity. The bits in an operand are recognized to have the following functions:

Bits	Function
[0:1]	Flag bit (=0)
[1:1]	Sign of mantissa (0=+, 1=-)
[2:1]	Sign of exponent (0=+, 1=-)
[3:6]	Exponent
[9:39]	Mantissa

The decimal point is assumed to be at the extreme right for fixed and floating point numbers. Fixed point numbers (i.e., integers) have a zero exponent.

Data Descriptors

A data descriptor, as its name implies, describes data (i.e., a data area) by pointing to one or more contiguous data locations. Consequently, a particular data descriptor may be concerned with many memory

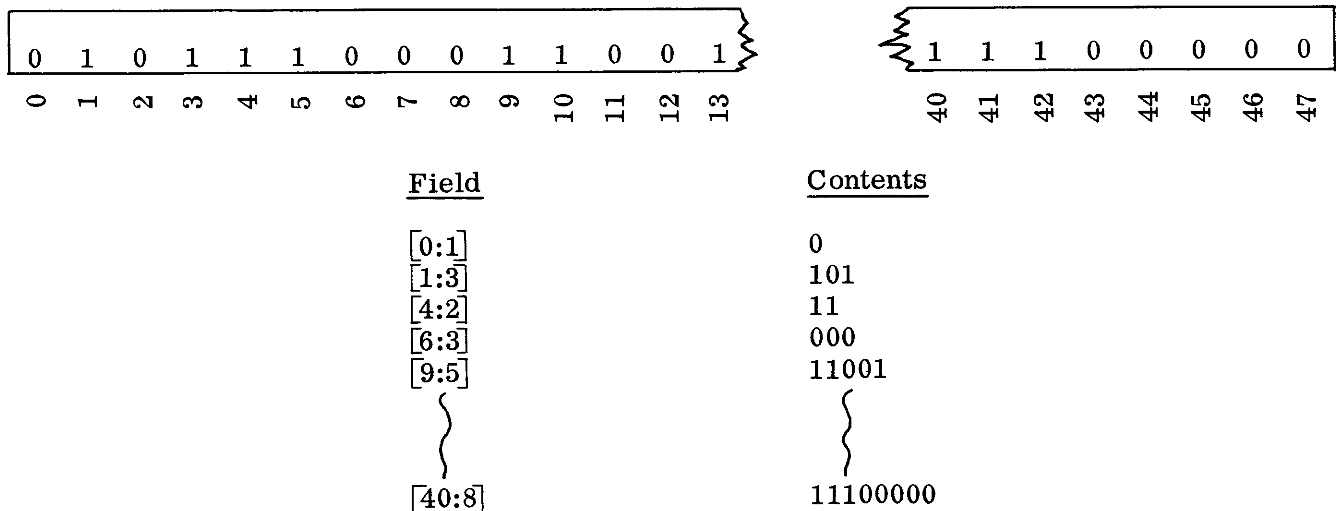


Figure 4. Bit Reference

locations. More than this, however, a data descriptor is also concerned with the presence, in core, of the data it describes. This is necessarily so particularly because of the data overlay capabilities of the B 5500. A descriptor is concerned with many aspects of storage. These aspects are indicated by various bits in the descriptor. The bits in a data descriptor have the following functions:

Bits	Function
[0:1]	Flag bit (=1)
[1:1]	Identification bit (=0, for data descriptor)
[2:1]	Presence bit (1=present, 0=not present)
[3:5]	Reserved for system use
[8:10]	Size Field
[18:15]	Reserved for system use
[33:15]	Address of data in core, if present.
	Address of data in overlay storage, if overlaid.
	Code number[1], if the area has never been accessed.

Bit Explanation

IDENTIFICATION BIT. Descriptors are used for other functions as well as describing data; this bit set to zero signifies this to be a data descriptor.

PRESENCE BIT. As noted above, the data area represented by a particular data descriptor may or may not be present in core. If the data is not present in core, the DF MCP sets this bit to 0; if the data has been made present, the DF MCP sets this bit to 1.

[1]The code number specifies the type of storage required (e.g., overlayable or non-overlayable); this will be explained further in Section 3.

SIZE FIELD. This field denotes the number of words in the area represented by this descriptor.

ADDRESS FIELD. As shown above, this field may specify any one of three things. To discern what the field represents, the following logic may be used:

1. If the presence bit is 1, then the field represents a core address.
2. If the presence bit is 0, then
   a. if the field is a code number, then the area has never been obtained; otherwise,
   b. the field represents an address in overlay storage.

OPERAND CALL AND DATA WORDS

The function of the operand call syllable is to obtain an operand and place it in the top of the stack. The method by which this is accomplished varies, according to the kind of data word addressed.

Operand Call on an Operand

When an operand call addresses an operand, the operand is brought to the top of the stack; the processor automatically checks the flag bit and, finding it 0, terminates the operand call operation leaving the operand in the top of the stack.

Operand Call on a Data Descriptor

When an operand call addresses a data descriptor, the data descriptor is brought to the top of the stack; the processor automatically checks the flag bit and finding it 1 continues the operand call operation (i.e., the operation as it is related to data descriptors) as follows:

1. The presence bit of the descriptor is checked and
   a. if it is 0, the processor's presence bit interrupt is set, an interrupt occurs, and the DF MCP assumes control (brings the data to core).
   b. if it is 1, the processor continues as shown at 2.

2. The size field is checked for 0 and

   a. if it is zero, the word addressed by the address field of the descriptor is placed in the top of the stack, replacing the descriptor. The word thus obtained should be an operand. If it were not, a "flag bit interrupt" would occur and the DF MCP would take control.

   b. if the size field is not zero, this indicates an index operation is requested. When this is the case, the second word from top word in the stack (i.e., the word in rB) is taken to be the index value.

It should again be pointed out that the operations described above show the paths which the processor would choose and follow automatically to complete the execution of an operand call syllable which accesses a data descriptor (figure 5).

Indexing

The processor automatically performs the index operation essentially as follows:

1. The index value in rB is compared to the size field in the data descriptor in rA and

   a. if it would index outside of the data area, the processor's invalid index interrupt is set, an interrupt occurs, and the DF MCP takes control.

   b. if it is a valid index, the processor continues as shown at 2.

2. The index value in rB is added to the address in the data descriptor (leaving only the indexed descriptor).

3. The word addressed by the indexed data descriptor is placed in the top of the stack replacing the descriptor. The word thus obtained should be an operand. If it were not, a "flag bit interrupt" would occur and the DF MCP would take control.

Figure 5 shows core conditions, and then the operations that take place during the execution of an operand call which addresses a data descriptor with a non-zero size field. The literal 7 is assumed in the top of the stack at start. rR = 34000.

POLISH NOTATION

The code[2] in programs for the B 5500 is in a form known as Polish Notation. Basically, Polish Notation is a method for writing expressions without a need for bracket characters to delimit the scope of an operator.

Conventional notation, as is well known, does rely on bracket characters for this purpose. For example, consider the different meanings of the two expressions noted below which only differ in the use of parentheses.

1. (6+9) / 3 (which equals 5)
2. 6 + 9 / 3 (which equals 9)

Polish notation will be described below through use of examples. The examples are designed to relate to computer operations and will deal only with evaluating arithmetic expressions and setting variables to values of those expressions.

The following notation will be used in the examples:

1. Variables will be represented by identifiers, where an identifier is an upper case letter or a contiguous string of upper case letters and digits, beginning with an upper case letter.

2. Where the value of a variable is represented, the variable will be subscripted with a lower case v. For example, Yv or Xv. Variables written in this manner are considered to be a form of operand.

3. Where the location of a variable is represented, the variable will be subscripted with a lower case a. For example Ya or Xa. This construct will be referred to as an address variable.

[2] The code referred to here is the machine code generated by the compilers for the B 5500 and should not be confused with the "code" written by programmers using the problem oriented compilers.

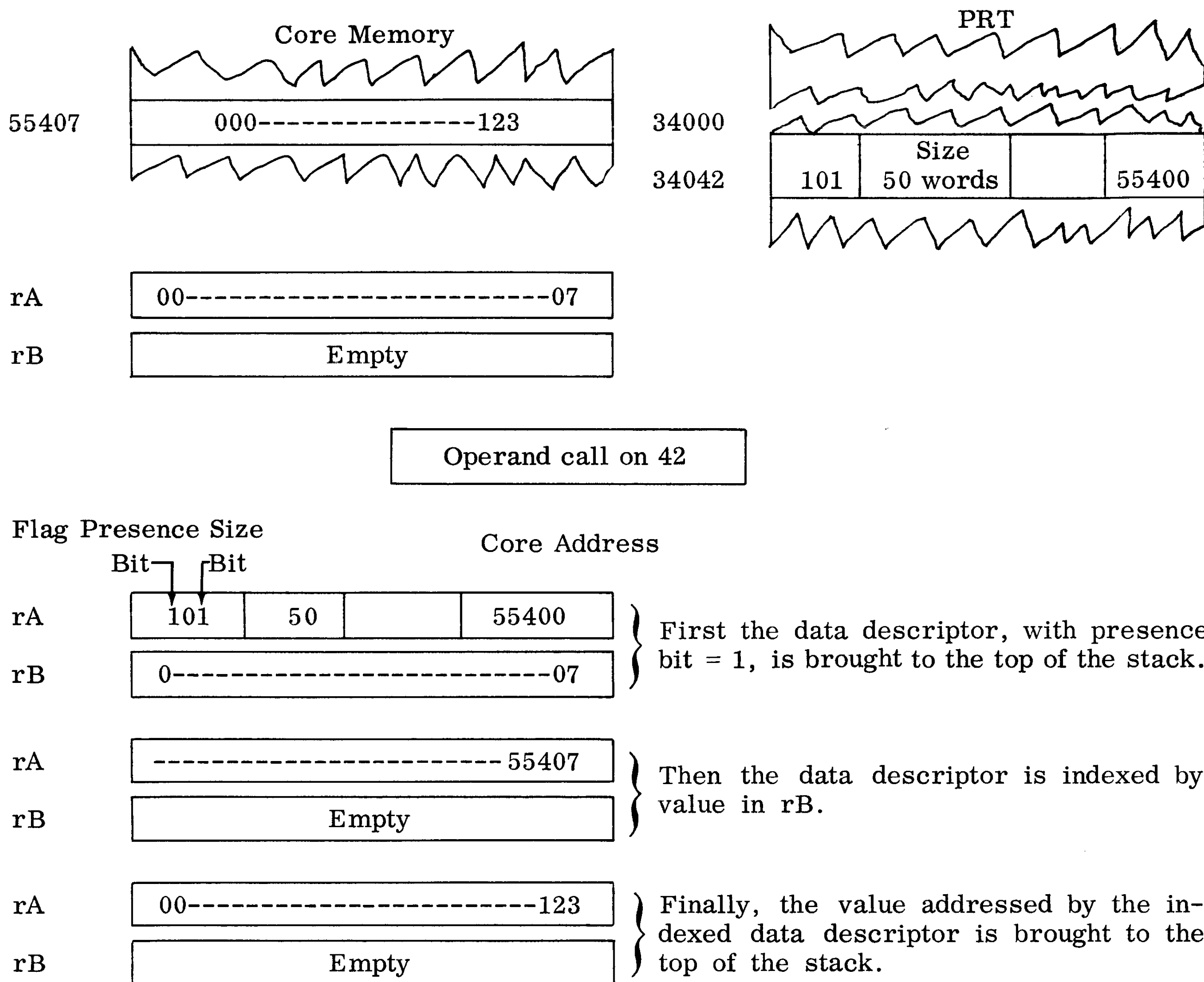


Figure 5. Operand Call Accessing a Data Descriptor

The following rules are used to interpret the Polish Notation used in the examples:

1. The Polish "string" is read from left to right.

2. Operands are obtained (i.e., read) until the last operand has occurred or until an arithmetic operator occurs.

3. When an arithmetic operator occurs, the two most recently obtained operands are operated upon as though the operator had occurred following the second from last operand and preceding the most recent operand.

4. When an arithmetic operation is performed on two operands, the result is one operand.

5. The sequence noted above continues until an address variable occurs.

6. When an address variable occurs, it will be followed by an assignment operator (i.e., the symbol ←). The occurrence of an address variable and an ← means that the address variable is to be assigned the value of the recently obtained operand (i.e., the value of the expression).

Extended ALGOL statements will be used with each example to express, in a more conventional manner, the operations to be performed.

The arithmetic operators used in the examples have their conventional meaning, as they do in Extended ALGOL. The Extended ALGOL symbol ← is referred to as the assignment symbol. It symbolizes that the value of the expression on its right is to be assigned to the variable on its left. For example, the statement.

X ← B + C

means that the value of B plus C is to be assigned to the variable X.

Examples of Polish Notation

Consider the Extended ALGOL statement:

X ← Y + Z + W

In Polish Notation the statement would be written:

Yv Zv + Wv + Xa ←

which in words means:

Obtain the value of Y.

Obtain the value of Z.

Add the two most recently obtained operands, (i.e., Y and Z).

Obtain the value of W.

Add the two most recently obtained operands (i.e., (Y+Z) and W).

Assign X the value of the most recently obtained operand (i.e., (Y+Z+W).

Consider the Extended ALGOL statement (there are four operand variables and VOL is the address variable):

VOL←QT x CON1 + PT x CON2

In Polish Notation this statement would be written:

QTv CON1v x PTv CON2v x + VOL←

which in words means:

Obtain the value of QT.

Obtain the value of CON1.

Multiply the two most recently obtained operands (i.e., QT and CON1).

Obtain the value of PT.

Obtain the value of CON2.

Multiply the two most recently obtained operands (i.e., PT and CON2).

Add the two most recently obtained operands (i.e., (QT x CON1) and (PT x CON2)).

Assign VOL the value of the most recently obtained operand (i.e., (QT x CON1 + PT x CON2)).

Consider the Extended ALGOL statement:

X←(D x (M + N)) / (T + P)

In Polish Notation the statement would be written:

DV Mv Nv + x Tv Pv + / Xa←

which in words means:

Obtain the value of D.

Obtain the value of M.

Obtain the value of N.

Add the two most recently obtained operands (i.e., M and N).

Multiply the two most recently obtained operands (i.e., (M + N) and D).

Obtain the value of T.

Obtain the value of P.

Add the two most recently obtained operands (i.e., T and P).

Divide the next to last obtained operand by the most recently obtained operand (i.e., (D x (M + N) by (T + P)).

Assign X the value of the most recently obtained operand (i.e., ((D x (M + N) / (T + P)).

SYLLABLES: INSTRUCTIONS FOR THE B 5500

Syllables have been discussed in the preceding paragraphs; namely the operand call syllable and an operator syllable, the store operator. In all, the B 5500 recognizes four types of word mode syllables: the operand call syllable, the descriptor call syllable, the literal call syllable, and the operator syllable.

A syllable is 12 bits in length, thus accounting for the fact that one B 5500 word accommodates four syllables. The two right-most bits in a syllable disclose the type of syllable; the remaining ten bits vary in use, depending upon the type of syllable.

Operand Call Syllable (10)

The two right-most bits of the operand call syllable are 10. The ten remaining bits contain a relative address. The operand call was discussed previously.

Descriptor Call Syllable (11)

The two right-most bits of the descriptor call syllable are 11. The ten remaining bits contain a relative address. The action of the descriptor call is very similar to the operand call. The function of the descriptor call, however, is to bring a descriptor into the top of the stack.

Literal Call Syllable (00)

The two right-most bits of the literal call syllable are 00. The ten remaining bits are a literal value from 0 to 1023. The execution of a literal call syllable causes the literal value contained in the syllable to be placed in the top of the stack as a positive integer.

Operator Syllable (01)

The two right-most bits of an operator syllable are 01. The ten remaining bits specify the function of the operator (e.g., there is a single precision add operator, a single precision subtract operator, etc.). Operator syllables designate the manner in which the data in rA and/or rB (i.e., the top stack locations) are to be operated on; or, in the case of double precision operators, how the data in the top four stack locations are to be operated on.

POLISH NOTATION AND B 5500 PROGRAMS

As was noted above, the machine language code of programs for the B 5500 is a form of Polish Notation. This is possible primarily because of the stack facility of the B 5500.

To illustrate the importance of the stack, first reconsider its characteristics.

1. When information is placed in and used from the stack, it is always such that the most recently obtained information is in the top of the stack.

2. The arithmetic registers, rA and rB, are always essentially the top words in the stack.

3. The registers rA and rB are always kept full automatically, as needed.

Second, reconsider how the operand call and literal call syllables bring values to the top of the stack; and how the storing operators use the stack.

Third, consider these single precision arithmetic operators of the B 5500:

1. The ADD operator causes the value in rA to be added to the value in rB. The sum is left in rB and rA is marked empty.

2. The SUBTRACT operator causes the value in rA to be subtracted from the value in rB. The difference is left in rB and rA is marked empty.

3. The MULTIPLY operator causes the value in rB to be multiplied by the value in rA. The product is left in rB and rA is marked empty.

4. The DIVIDE operator causes the value in rB to be divided by the value in rA. The quotient is left in rB and rA is marked empty.

Fourth, recognize the following conventions:

1. OPDC (variable) represents an operand call syllable containing the relative address of the indicated variable. For example:

   OPDC (P1)

   represents an operand call syllable that would place the value of P1 in the top of the stack.

2. LITC (literal) represents a literal call syllable which contains the literal value indicated. For example:

   LITC (14)

   would cause a positive integer 14 to be placed in the top of the stack.

3. LITC (address of variable) represents a literal call syllable which contains a literal value equal to the relative address of the indicated variable. For example:

   LITC (address of X)

   would cause a literal equal to the relative address of X to be placed in the top of the stack.

4. ADD represents the single precision add operator.

5. SUB represents the single precision subtract operator.

6. MUL represents the single precision multiply operator.

7. DVS represents the single precision divide operator.

8. STD represents the store destructive operator. This operator will, if rA contains an operand, use the literal value in rA as a relative address, and store the value in rB in the location relatively addressed by rA. After the storing, rA and rB are marked empty.

Now again, consider the examples of Polish Notation together with equivalent Extended ALGOL statements; however, this time augmented by equivalent B 5500 code, represented mnemonically.

The B 5500 code in the examples will be listed in sequence, by syllable. To the right of each syllable, notation will be shown that would be the contents of rA and rB after the execution of that syllable. Registers rA and rB will be assumed empty at the beginning of the code.

PROGRAMS FOR THE B 5500: SEGMENTED AND ADDRESS INDEPENDENT

A program for the B 5500 is made up of program segments. Program segments have the following characteristics: (1) the only segment that need be in core at a particular time is the one being executed. (2) a segment is address-independent and therefore not dependent upon where it is placed in core.

The principal B 5500 features that make address-independent segments possible are the C register and relative branching. A special function of the operand call syllable facilitates the handling of non-present segments.

C Register: Address of Control

The C register is used to contain the core address of the program word that contains the syllable to be executed next. Disregarding any branching actions, the B 5500, in regard to rC, functions essentially as follows.

> The word addressed by rC is placed in register rP. Individual syllables are taken from rP, in sequence, and executed. As the last syllable from rP is executed, rC is incremented and the next program word is placed in rP, etc. The L register specifies the next syllable in rP to be executed. Syllables are numbered 0, 1, 2, 3 from left to right.

Examples of Statements and Resulting Code

Example 1.

The Extended ALGOL statement:

X ← Y + Z + W

is represented in Polish Notation as:

Yv Zv + Wv + Xa ←

The equivalent B 5500 code, expressed mnemonically, and the contents of rA and rB are as follows:

OPDC (Y)	rA = Y	rB = empty
OPDC (Z)	rA = Z	rB = Y
ADD	rA = empty	rB = Y + Z
OPDC (W)	rA = W	rB = Y + Z
ADD	rA = empty	rB = Y + Z + W
LITC (address of X)	rA = address of X	rB = Y + Z + W
STD[3]	rA = empty	rB = empty

Example 2.

The Extended ALGOL statement:

VOL ← QT x CON1 + PT x CON2

is represented in Polish Notation as:

QTv CON1v x PTv CON2v x + VOLa ←

The equivalent B 5500 code, expressed mnemonically, and the contents of rA and rB are as follows:

OPDC (QT)	rA = QT	rB = empty
OPDC (CON1)	rA = CON1	rB = QT
MUL	rA = empty	rB = QT x CON1
OPDC (PT)	rA = PT	rB = QT x CON1
OPDC (CON2)	rA = CON2	rB = PT
MUL	rA = empty	rB = PT x CON2
ADD[4]	rA = empty	rB = QT x CON1 + PT x CON2
LITC (address of VOL)	rA = address of VOL	rB = QT x CON1 + PT x CON2
STD[5]	rA = empty	rB empty

[3] Location of X = Y + Z + W

[4] Immediately before the addition, the stack was automatically adjusted so rA = PT x CON2 and rB = QT x CON1.

[5] Location or VOL = QT x CON1 + PT x CON2

Example 3.

The Extended ALGOL statement:

$$X \leftarrow (D \times (M + N)) \ / \ (T + P)$$

is represented in Polish Notation as:

$$Dv \ Mv \ Nv + x \ Tv \ Pv \ + / \ Xa \leftarrow$$

The equivalent B 5500 code, expressed mnemonically, and the contents of rA and rB are as follows:

OPDC (D)	rA = D	rB = empty
OPDC (M)	rA = M	rB = D
OPDC (N)	rA = N	rB = M
ADD	rA = empty	rB = M + N
MUL[6]	rA = empty	rB = D x (M + N)
OPDC (T)	rA = T	rB = D x (M + N)
OPDC (P)	rA = P	rB = T
ADD	rA = empty	rB = T + P
DVS[7]	rA = empty	rB = (D x (M + N)) / (T + P)
LITC (address of X)	rA = address of X	rB = (D x (M + N)) / (T + P)
STD[8]	rA = empty	rB = empty

When a particular segment is to be executed, the DF MCP can cause rC to be set to the proper core address, according to where the program segment was placed in core.

Transferring Control within Segments: Relative Branching

Syllables within a program segment are executed in sequence, except for transfers of control. To transfer control from one syllable in a program to another in the same segment, no syllable addresses are used. Transferring control is done by specifying: (1) the number of syllables or words the branch is to span (calculated in relation to the current point of execution), and (2) the direction of the branch, either forward or backward. For example, a particular transfer of control could specify to "branch forward 15 syllables".

It should be noted that this manner of branching does not rely on the program segment to be in specific core locations. Considering this fact, and the fact that the DF MCP can determine the setting of rC, it can easily be seen that program segments may be located at the discretion of the DF MCP.

Transferring Control to Other Segments and the Operand Call

The B 5500 branch instructions are designed to execute one of two ways: (1) if there is a literal in the top of the stack when the branch is executed, then it is to be executed as a relative branch and the literal specifies the number of syllables (or words, depending on

---

[6] Immediately before the multiplication, the stack was automatically adjusted so rA = M + N and rB = D.

[7] Immediately before the division, the stack was automatically adjusted so that rA = T +P and rB = D x (M+N).

[8] Location X = (D x (M+N)) / (T+P).

the branch instruction) that the branch is to span, as described above; (2) if there is a descriptor in the top of the stack, then the branch is to transfer control to the absolute address in the descriptor.

To transfer control from one program segment to another, a branch is executed with a descriptor, in the top of the stack, that addresses the segment to which control is to be transferred. This method of transferring control is straight forward, except for one factor; the segment being branched to may not be in core. The segments presence must be ensured.

The branching descriptor, which is placed in the top of the stack, is a special form of a program descriptor referred to as a Label Descriptor. A Label Descriptor is described as follows:

Bits	Function
[0:1]	Flag bit (=1)
[1:1]	Identification bit (=1, for program descriptor)
[2:1]	Presence bit (1 = present, 0 = not present)
[3:3]	Special function (=110, for label descriptor)
[6:27]	Reserved for system use
[33:15]	Address of program segment in core, if present.

When an operand call addresses a label descriptor, the label descriptor is brought to the top of the stack. The processor finds the flag bit to be 1, and checks the presence bit. If the presence bit is 0, the processor's presence bit interrupt is set and the DF MCP takes control. If the presence bit is 1, the processor checks to see if the descriptor is a label descriptor. When it is found to be a label descriptor, the descriptor is left, as is, in the top of the stack and the operand call operation is terminated.

The special way in which the operand call syllable treats a label descriptor is as follows:

1. An operand call addressing a label descriptor is executed prior to a branch which will cause control to be transferred to a different segment and one of two results will occur:

    a. If the segment is present, then the presence bit of the label descriptor will be 1, the label descriptor will be left in the top of the stack, and the branch instruction will be executed.

    b. If the segment is not present, control will go to the DF MCP. The DF MCP will make the segment present, place the new address of the segment in the label descriptor, do all necessary fix-up, and return control to the program which can then branch to the new segment.

PROCEDURES: SUBROUTINES OF THE B 5500

A B 5500 subroutine, here referred to as a procedure, has three principal constituents: (1) a code string, (2) parameters, and (3) local variables. The code string can be branched to from various points in a program by making a "call" on the procedure. When the procedure is completed, program control will be returned to the point following the procedure call.

When a procedure is called, parameters, if any, are placed in the program stack. Variables used by the procedure, and which are local to the procedure, are assigned stack locations. The procedure can reference the parameters and variables by use of relative addresses.

The B 5500 procedure facility is related principally to three B 5500 features: the F register, control words, and a type of program descriptor called the procedure descriptor.

F Register

The F register is used to contain an address of reference for relative addresses of stack locations. The F register also serves to mark the locations of control words so that: (1) their addresses can be recorded in subsequent control words when entering a procedure, or (2) their contents can be used when exiting a procedure.

Control Words for Procedures

Control words are automatically created by the processor when certain operators are executed or when interrupts occur. They contain the contents of various registers and other such information which can be used to re-establish processing conditions after the occurrence of subroutines and interrupts. The two control words used with procedure handling are: (1) the mark stack control word and (2) the return control word.

MARK STACK CONTROL WORD. Included in the information contained in the mark stack control word is the value of rF when the mark stack control word was being created. The mark stack control word precedes procedure parameters.

RETURN CONTROL WORD. Included in the information contained in the return control word are: (1) the value of rF, (2) the value of rC, and (3) the value of rL when the return control word was being created. The return control word follows procedure parameters.

Procedure Descriptor

The procedure descriptor is a program descriptor uniquely marked as a descriptor which will cause subroutine entry when addressed by an operand call syllable.

The information in a procedure descriptor includes the following:

Bits	Function
[0:1]	Flag bit (=1)
[1:1]	Identification bit (=1, for program descriptor)
[2:1]	Presence bit (1 = present, 0 = not present)
[3:3]	Special function (=101, for procedure descriptor)
[6:27]	Reserved for system use
[33:15]	Address of procedure code in core, if present

Calling a Procedure

When calling a procedure, the first syllable to be executed is the mark stack operator.

MARK STACK OPERATOR. Actions caused by the mark stack operator include the following.

1. The contents of rA and rB, if any, are pushed into the core memory portion of the stack.

2. A mark stack control word is constructed in rB and pushed into the core memory portion of the stack. If this mark stack control word is the first preceding a return control word, the mark stack control word is also placed in the PRT at rR + 7.

3. The F register is set to the address of the top of the stack (i.e., rF is set to the current value of rS which addresses the location containing the mark stack control word).

4. The processor is put in the sub-program level, if it had not been previously.

After the mark stack operation, the program can place parameters in the stack, if desired. Parameters would include operands and/or descriptors. Then, an operand call addressing the pertinent procedure descriptor would be executed.

OPERAND CALL ON A PROCEDURE DESCRIPTOR. Actions caused by executing an operand call that addresses a procedure descriptor include the following:

1. The descriptor is placed in rA and found to have a flag bit of 1.

2. The presence bit is checked. If 0, a presence bit interrupt occurs and the DF MCP takes control to make the procedure code present. If 1, step 3 would follow immediately.

3. The descriptor is determined to be a program descriptor.

4. If there is a word in rB, it is pushed into the core memory portion of the stack.

5. A return control word is constructed in rB, and pushed into the core memory portion of the stack.

6. The C register is set to the address in the procedure descriptor, (i.e., the address of the procedure code in core).

7. The F register is set to the address of the top of the stack (i.e., rF is set to the value of rS, which currently addresses the location containing the return control word).

At the completion of the operand call syllable, the processor begins executing the procedure because rC, at that time, addresses the procedure code.

Certain conditions existing in the stack should be noted:

1. The mark stack control word is the "deepest" word in the stack that pertains to the procedure. It contains the value contained by rF prior to the procedure call.

2. The parameters to the procedure follow the mark stack control word and, more important, immediately precede the return control word.

3. The value of rF that was placed in the return control word is the address of the mark stack control word.

4. The value of rC that was placed in the return control word addresses the program word to which control can be returned at the completion of the procedure. Register rL specifies the syllable within the program word.

5. The F register contains the address of the return control word.

Executing Procedure

When a procedure begins execution, the first action is to reserve stack locations for its local variables. This can be done by executing literal call syllables containing a zero value; one such literal call for each local variable. This places zeros in the stack immediately following the return control word which is addressed by rF (rS is adjusted if necessary). An important factor then is how the stack locations which contain the parameters and local variables can be addressed.

RELATIVE ADDRESSING IN THE SUB-PROGRAM LEVEL. When operating in the sub-program level, the B 5500 can do addressing relative to rR and rF. In the sub-program level, addressing may also be done relative to rC and the value addressed by rR + 7; however, this will not be discussed. When addressing relative to rR, the relative addresses are always added to the value of rR. When addressing relative to rF, relative addresses may be added to or subtracted from the value of rF. The conditions stated below dictate whether addressing is to be rR-plus, rF-plus, or rF-minus. The setting of a 1-bit machine register called the mark stack flip-flop is also related to addressing in sub-program level. For this discussion it is assumed to be zero.

1. If the left-most bit of an operand call syllable or descriptor call syllable is 0, the relative address (i.e., the remaining nine bits in the address portion of the syllable) will be added to the value in rR.

2. If the two left-most bits of an operand call syllable or descriptor call syllable are 10, the relative address (i.e., the remaining eight bits in the address portion of the syllable) will be added to the value in rF.

3. If the three left-most bits of an operand call syllable or descriptors are 111, the relative address (i.e., the remaining seven bits in the address portion of the syllable) will be subtracted from the value in rF.

4. If a store operator is executed and the left-most bit of the relative address in rA is zero, the remaining nine bits in the address will be added to rR.

5. If a store operator is executed and the two left-most bits of the relative address in rA are 10, the remaining eight bits of the address will be added to rF.

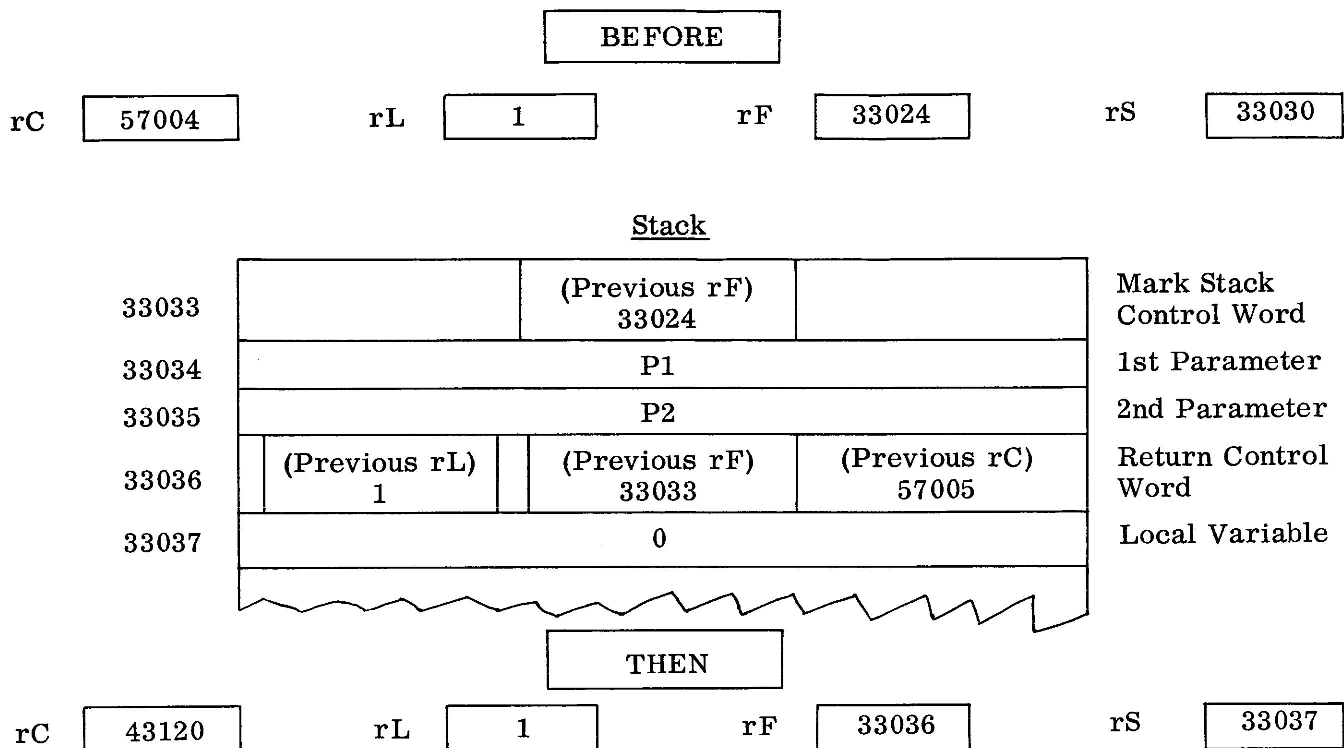


Figure 6. Sub-Program Level Relative Addressing

6. If a store operator is executed and the three left-most bits of the relative address in rA are 111, the remaining seven bits of the address will be subtracted from rF.

The absolute addresses, obtained as specified above, are placed in rM. The pertinent values are then obtained (or stored) in the conventional manner, as was discussed above in relation to the subject syllables.

Considering the conventions for addressing noted above and remembering that rF points at the return control word, it can be seen that a procedure can reference its parameters by using rF-minus addressing, its local variables by using rF-plus addressing, and PRT variables by rR-plus addressing.

Figure 6 shows: (1) rC, rL, rF, and rS before a procedure call; (2) the stack arrangement after a procedure call and after one location was reserved for a local variable. (The procedure call required the execution of four syllables: (a) a mark stack operation, (b) two operand calls for parameters, P1 and P2, and (c) an operand call on a procedure descriptor with an address field of 43120); and (3) registers rC, rF, and rS after the stack was set up as shown. Register rA and rB were <u>full</u> before the procedure call.

Exiting a Procedure

When a procedure is exited, the following is essentially what automatically occurs:

1. The word addressed by rF is placed in rB (i.e., the return control word is placed in rB).

2. The registers rC and rL are restored to the values contained in the return control word (i.e., rC and rL are set to the point in the program which follows the procedure call).

3. Other registers whose settings are stored in the return control word are restored, excluding rF.

4. Register rS is set to the value of rF.

5. The word addressed by rS (i.e., the mark stack control word) is placed in rB.

6. The registers whose settings are stored in the mark stack control word (in rB) are restored, including rF.

7. Register rS is decremented by 1, thus restoring it to the setting it contained before entering the procedure.

8. Control is returned to the point designated by rC and rL.

Additional Remarks about Subroutines

There are additional features and special conditions related to subroutines on the B 5500 that are not discussed here. The information above, however, does show the general pattern for handling subroutines and is sufficient for the purpose of this document.

Two remaining points that should be added are:

1. Procedures which bring back values (e.g., functions) accomplish this by placing the value in rA before going through the exiting process discussed above.

2. Procedures can call on other procedures and/or themselves. The process for doing this is the same as the process discussed above. The only differences would be in register settings.

B 5500 PROCESSORS AND STATES

The B 5500 can utilize one or two processor units. Processor units can be designated Processor 1 or Processor 2 by means of a hardware switch. However, if there is only one processor unit on a system, it must be designated Processor 1; and if there are two processors, one must be designated Processor 1 and the other Processor 2.

A B 5500 processor can operate in either of two "states", control state or normal state. However, the processor designated processor 2 can operate only in normal state.

All available memory modules may be used by either processor; however, the first 512 words of memory module zero are reserved for control state use and are consequently available only to processor 1.

Normal State

When operating in normal state, a processor can make use of all operators necessary for computation. Therefore, the execution of explicit object program code is performed in normal state. When two processors are available on a B 5500 System, both processors can simultaneously execute object programs.

Processing in normal state can be interrupted when an interrupt condition is detected. Interrupt conditions are handled in control state.

Control State

The DF MCP controls operations in control state. When operating in control state, a processor can make use of an extended set of operators. The additional operators are needed to perform operations required of the control program which is the DF MCP. If special control state operators are encountered during normal state processing, they are treated as no-ops.

Processing in control state is not interrupted when an interrupt condition is detected. Certain interrupt conditions such as "presence bit" are inhibited. Other interrupt conditions such as those caused by "I/O finished" signals are recorded, but only recognized when the DF MCP does an "interrogate interrupt" operation (this operation will be discussed below).

INTERRUPTS

There are a number of conditions recognized by the B 5500 as interrupt conditions. Each interrupt condition is associated with a bit (or unique combination of bits) in an interrupt

register. In turn, each bit (or bit combination) in the interrupt register is associated with a given location in core referred to as the interrupt location for the particular interrupt.

Processor Dependent and Processor Independent Interrupts

Interrupt conditions can be generated by a processor or by other components of the hardware. Interrupt conditions generated by a processor are called processor dependent; those generated by other components of the hardware are called processor independent.

An example of a processor dependent interrupt condition is the "presence bit condition" caused by a program being executed on a processor which is executing an operand call which addresses a descriptor with a presence bit of zero.

An example of a processor independent interrupt condition is an "I/O finished condition" caused by the I/O hardware when an I/O operation has been completed.

Processor dependent interrupts are classed as Processor 1 interrupts and Processor 2 interrupts. That is, there is a particular interrupt bit associated with each interrupt caused by Processor 1 and likewise for Processor 2. The types of interrupt that may be caused on either processor are the same.

When Processor 2 is operating in normal state, its operation can be interrupted only by the occurrence of a Processor 2 interrupt condition or HALT P2 instruction executed by Processor 1.

When Processor 1 is operating in normal state, its operation will be interrupted by the occurrence of <u>any</u> interrupt condition.

Action to Handle Interrupts

When a processor is operating in normal state and its operation is interrupted, the following actions occur:

1. The contents of rA and rB, if any, are pushed into the core memory portion of the stack.
2. An interrupt control word is constructed and pushed into the core memory portion of the stack.
3. An interrupt return control word is constructed and pushed into the core memory portion of the stack.
4. An initiate control word is constructed and placed in the program's PRT at rR + 8. The value of rS in this word addresses the interrupt return control word.

INTERRUPT CONTROL WORD. Included in the contents of the interrupt control word are the contents of rR and the contents of rM at interrupt time.

INTERRUPT RETURN CONTROL WORD. Included in the contents of the interrupt return control word are the contents of rC, the contents of rL, and the contents of rF at interrupt time.

INITIATE CONTROL WORD. Included in the contents of the initiate control word is a bit that specifies whether the interrupt occurred during a word mode operation or a character mode operation, and the address of the top of the stack when control is switched to the DF MCP.

If the above actions occur on Processor 2, an appropriate interrupt bit for that processor is set and the processor idles until it is re-initiated by Processor 1. The interrupt bit set by Processor 2 causes Processor 1 to be interrupted if Processor 1 is in normal state.

After the above actions occur on Processor 1, the following actions occur:

1. The processor goes into control state.
2. The C register is set to the core address of an interrupt location (rL is set to zero).
3. The S register is set to memory address 64, thus relating rA and rB to the first words of the area which can now be used as a DF MCP stack. This area is in the portion of core reserved for control state.

4. The DF MCP processes the interrupt(s).

5. The DF MCP determines that a program can be re-initiated.

6. An initiate control word is taken from a PRT and placed in rA.

7. The DF MCP executes an "Initiate Processor 1" syllable (or "Initiate Processor 2") to put operations back to normal state. The remaining steps occur automatically.

8. The S register is automatically set from the value in the initiate control word in rA; thus rA and rB are again related to the program's stack.

9. The word addressed by rS (the interrupt return control word) is read and the address values therein are placed in their respective registers.

10. The S register is decremented by 1 so that it addresses the interrupt control word.

11. The interrupt control word is read and its address values are distributed to their respective registers.

12. The S register is left as it was before the interrupt and the normal state processing continues from the point of interrupt.

It should be noted that more than one interrupt condition can exist at a time. For instance, one or more "I/O complete" interrupts (depending on the number of I/O control units) could occur while the DF MCP was in control state handling a presence bit interrupt. It would be inefficient if the DF MCP re-established conditions for, and returned operations to, normal state after one interrupt only to have another interrupt recognized. The interrogate interrupt instruction allows for this situation to be avoided.

Interrogate Interrupt Operator

The interrogate interrupt instruction is a special control state operator. When it is executed, interrupt bits are interrogated on a priority basis. If an interrupt bit is on, actions happen much the same as when an interrupt occurs in normal state (control words excluded). Functionally:

1. The C register is set to a core address reserved for the particular interrupt (rL is set to zero).

2. The S register is set to core address 64.

3. The DF MCP processes the interrupt. (Note: When an interrupt is handled, the pertinent interrupt bit is then automatically turned off.)

If no interrupt bits are on when the interrogate interrupt is executed, the instruction acts like a no-op.

The DF MCP always interrogates all interrupts before returning Processor 1 to normal state.

I/O ON THE B 5500

I/O operations on the B 5500 are processor initiated, but I/O control units (often referred to as I/O channels), which perform independently of the processor, control the transfer of data between memory and peripheral units.

Initiating an I/O

I/O operations can be initiated only by Processor 1 operating in control state. To cause an I/O operation, the address of an I/O descriptor is placed in rA and then an initiate I/O (IIO) operator is executed. The IIO causes the contents of rA to be stored in memory location 8 (marking rA empty) and then causes an IIO signal to be sent to computer's central control. Upon receiving an IIO signal, the computer's central control seeks an available I/O control unit. The B 5500 may have from one to four I/O control units. Any I/O control unit may control I/O for any peripheral unit. Selection of I/O control units is done automatically on the basis of availability. When an I/O control unit is selected, central control causes the contents of memory location 8 to be transferred to the unit. From this point on, processing continues and the I/O is handled independently of the processor.

I/O Descriptor

Each peripheral unit for the B 5500 has a number associated with it, referred to as the unit designation. Each number is recognized by the hardware to be associated with a particular unit. An I/O descriptor contains parameters specifying information needed to perform an I/O operation. The parameters specify the unit designation of the unit which is to perform the I/O, the type of I/O, whether read or write, the initial memory address where data is to be written or read, and the number of words to be transferred, etc.

Operation of an I/O Control Unit

When an I/O control unit receives the address of an I/O descriptor, the descriptor is fetched and placed in a register within the I/O control unit. Then the operation of transferring data begins as specified by the I/O descriptor, except in the case of disk I/O.

In the case of disk I/O, the memory address in the I/O descriptor is not that of the data area, but rather the address of the first word preceding the data area. This is so because for disk I/O, the first word preceding the data area must contain the disk address involved in the operation.

When an I/O control unit receives an I/O descriptor with a unit designation for disk, the address word preceding the data area is fetched and placed in a register within a disk file control unit. Then the operation of transferring data begins under control of both the I/O control unit and the disk file control unit.

I/O Completion

During the time a peripheral unit is involved in an I/O operation, it is not available for additional operations. Therefore, there must be a way for the processor to know when an I/O control unit has completed the operation initiated for a particular peripheral unit. If during the I/O operation an error should occur, such as the recognition of invalid characters when reading a card or parity errors when reading or writing tape, there must be a way to supply the processor with error information.

To allow processor notification of I/O completion, the B 5500 has an I/O finish interrupt associated with each I/O control unit. To provide I/O result information such as error information, the B 5500 has a reserved memory location associated with each I/O control unit. (Core locations 12 through 15 are reserved for I/O control units 1 through 4 respectively.) When an I/O control unit completes an I/O (or when an I/O control unit finds the designated unit unavailable), two actions occur; an I/O result descriptor generated by the I/O control unit is stored in the control unit's reserved memory location, and the I/O finish interrupt for the control unit is set.

I/O Result Descriptor

A result descriptor is a word generated by an I/O control unit to provide information to the processor about the I/O operation just performed. The result descriptor contains bits that specify whether certain errors occurred, a bit that specifies if a unit was not available, and other information such as the memory address of the word following the last word accessed during a write operation.

Handling an I/O Finish Condition

When an I/O finish interrupt causes control to be transferred to an interrupt location, the DF MCP can then obtain the result descriptor in the memory location reserved for the I/O control unit that causes the interrupt. Then using the result descriptor information, the subsequent course of action can be decided.

A Note on Multiprocessing

From the information about interrupts, it can be noted that all the information required to re-initiate an interrupt program is stored in the program's PRT and core stack. It should be mentioned that, while a program is being processed, its PRT and core stack always occupy the same core area; that is, they are never overlaid. Also, certain program information, such as the code segment being executed when an interrupt occurs, is never overlaid. It can be surmised from this that more than one program could be processed by one processor during a given period of time.

Take for example a situation where two programs are being processed. Conditions might occur essentially as follows:

1. The DF MCP starts program 1.
2. Program 1 performs a number of I/O's causing all buffer areas to be in use and must wait for an I/O operation to be completed to "free" a buffer area.
3. The DF MCP starts program 2.
4. Program 2 gets interrupted and the DF MCP finds that program 2 must wait for a program segment to be made present.
5. The DF MCP handles an I/O finish and program 1 can resume processing.
6. The DF MCP obtains the initiate control word from the PRT of program 1, places it in rA, and initiates the processor so that program 1 continues.
7. Program 1 gets interrupted as in step 2.
8. The DF MCP finds program 2 is ready.
9. Program 2 is re-initiated as was program 1 in step 6.
10. etc.

Although there is more involved in handling multiprocessing than is spelled out above, it can be noted that the design of the B 5500 provides for handling of multiprocessing in a straight forward and efficient manner.

SECTION 2
THE DISK

GENERAL

This section encompasses the format and use of disk storage. Disk storage is divided into two categories: System Disk and User Disk. System Disk is the disk area reserved for: (1) the DF MCP program and tables, (2) the disk directory, (3) the available-disk list, (4) overlay storage, and other DF MCP uses. User disk is the area used for remaining facilities. Data files, scratch files, and library programs, including the B 5500 problem oriented compilers, may be stored in the user disk area.

USER DISK

Program Files vs. Data Files

Files on disk may, in terms of use, be classified as program files (i.e., library programs) and data files. The format of all files on disk, however, is basically the same. Consequently, user disk is not arbitrarily divided into fixed sections according to use; program files and data files may be intermixed. User disk is therefore divided only according to the demands of the user.

Format of Files on Disk

An area on disk to be used for a particular file must be explicitly reserved for that file. It follows from this that a program must specify the amount of disk required for a file. Some files are initially small, but in time grow large. In such cases, to reserve an explicit area of maximum size would result in the majority of the area initially lying idle. Also, disk storage may become "checker-boarded" due to the sequence in which file areas are assigned and returned. In these instances, the total area specified for a large new file might be available, but not contiguously. To reserve a large area explicitly, then, would require that existing files first be rearranged. To avoid such situations, the DF MCP allows a single file to occupy from one to twenty separate areas on disk. The number of areas and their size is specified by the program that creates the file. The fact that a file is stored in more than one area does not in any way affect the way it is referenced by a program. Regardless of the number of areas used, a program always addresses a file as though it were one continuous string of records.

SYSTEM DISK

Requirements

System disk is located in the first module of disk, starting at word zero. Excluding overlay storage, system disk requires a minimum of 1000 disk segments. One segment of disk is 240 characters in length, one disk module contains 40,000 segments.

Overlay Storage

Overlay storage is located in the high-order portion of system disk, immediately preceding user disk. Since overlay requirements may vary from one installation to another, the amount of storage reserved for overlay functions is not fixed. It may be determined by the user.

Disk Directory

The DF MCP maintains, on disk, a Disk Directory which provides information about

all permanent files on disk. The Disk Directory is composed of one or more "directory sections", depending upon the number of files on disk. Each directory section is composed of 16 segments and can contain the directory information required for as many as 15 files. The first segment of a directory section contains the names (i.e., file identifications) of each file defined in that section. The remaining 15 segments are referred to as file headers. There is one file header for each file defined in the section. Each file header contains various information about the file such as creation date, date of last access, etc. Each file header also specifies the number of areas declared for the file, the size of the areas, and the absolute disk address of each area. When a program is using a file, the file header for that file is read into core and remains there while the file is being used.

Available-Disk Table

The Available-Disk Table is a list containing an entry for each area of available disk storage. The list is composed of one or more segments, depending upon the number of available areas. Each segment contains from 0 to 29 entries. Each entry specifies the absolute address and size of an available disk area. The list is maintained in memory order (i.e., each list entry following the first entry defines an area with greater address).

INITIALIZING THE DISK

Two prerequisites to operation on a B 5500 Disk File System are: (1) the DF MCP must be on the disk and (2) a Disk Directory must be on the disk. To establish these prerequisite conditions, two "one-time routines" are provided to initialize the disk. A special loader routine reads the DF MCP from tape and places in on the disk. A second special routine places the initial Disk Directory in the directory area on disk. The second routine is required because the Disk Directory is considered to be a permanent record on disk. When directory information is needed, the DF MCP seeks the information, initially, by reading the first segment of the first section of the directory. This first section must be on disk to be read. The special routine creates a directory and places it on disk for the "cold start". Thereafter, the directory is maintained by the DF MCP which makes entries in the directory when permanent files are initially completed and removes entries upon notification.

SECTION 3

DISK FILE MASTER CONTROL PROGRAM

GENERAL

This section contains a discussion of the Disk File Master Control Program (DF MCP). The DF MCP is a program made up of a main body (the "outer block"), procedures, and tables (i.e., arrays). The outer block includes the code located at the B 5500's interrupt locations, a stack and PRT for the DF MCP, and a number of DF MCP routines. In the following discussion, coding in the outer block (which can gain control only through use of branching operation) is referred to as routine. Code which is entered through use of procedure descriptors is referred to as a "procedure".

INITIATING THE B 5500 DISK FILE SYSTEM

The B 5500 System is initiated when the machine operator performs a HALT-LOAD operation by pressing the HALT switch, then the LOAD switch. The HALT-LOAD operation automatically caused Processor 1 to go into control state and a portion of code to be read into the first locations of core memory in module zero. Control is then automatically transferred to core address 16 and the system is in operation. Initial operations cause the INITIALIZE Procedure and permanent segments of the DF MCP to be read from disk into core. The DF MCP then performs various initialization functions, including performing the first organization and classification of core storage, and creating the Available - Disk Table.

DF MCP CLASSIFICATION AND ORGANIZATION OF CORE STORAGE

As mentioned in Section 1, B 5500 programs are address-independent and consequently not restricted to particular areas of core. It was also mentioned that most information could be overlaid except such information as PRT and stack areas, and again made present when a presence bit interrupt indicated the need for the information. Two important points follow from these facts: (1) programs are not logically affected by the number of modules of core on a B 5500 System and (2) all available core storage on a B 5500 System can be used, overlaid, and reused in any fashion to meet the demands of all programs processing during a given period of time. It is evident that if core storage is to be put to use as indicated, it must be well classified and organized. Basically, storage is classified as non-overlayable, overlayable, or available, and is organized through use of "memory links".

Non-Overlayable Storage

There is a need for certain information to remain in core at all times. For example, the DF MCP has routines and tables that must frequently be used when handling interrupt conditions and other control functions. The space that would be momentarily gained by overlaying such information would not be worth the time required to make the information present when needed again.

There is also a need for certain object program information to remain in fixed locations while a program is being processed. This requirement holds for all information which will be referenced through use of absolute addresses. For example, control words contain absolute addresses of stack locations and program segments; also, the DF MCP keeps absolute addresses to use to locate PRT areas. From this it can be seen that

stack and PRT areas and in-use program segments must remain fixed for periods of time. The DF MCP classifies core areas containing information which must remain in place as non-overlayable storage.

Overlayable Storage

It is often the case that all information pertaining to a program cannot be in core at the same time. This is most often the case when multiprocessing and/or operating on systems with less than maximum core. However, the majority of information related to object programs, and most information in the DF MCP, may be used relatively infrequently and is never referenced through use of absolute addresses. In regard to such information as this, there is principally only one major factor determining its necessity to be present in core; it must be present when needed. The DF MCP classifies core areas containing information which need not remain in place as overlayable storage. There are, however, two classes of overlayable storage: (1) overlayable program segment areas and (2) overlayable data areas. It should be pointed out that these two types of areas must be handled somewhat differently.

OVERLAYABLE PROGRAM SEGMENT AREAS. B 5500 programs are always stored on disk during the time they are processing. Individual program segments are read into core as they are needed. An important feature about these program segments is that they are never programmatically modified. Consequently, if the area used by a program segment is to be overlaid, there is always an exact copy of it on disk. The DF MCP has only to mark the segment absent in appropriate places, and the area it occupied can be used for other purposes. If the segment is needed again, it can be read into core from disk.

OVERLAYABLE DATA AREAS. Data, unlike segments, is constantly subject to change by the object program. Consequently, if an area used for data (e.g., an array row) is to be overlaid, the data must first be written onto overlay storage. Then the DF MCP can mark it absent. If the data is needed again, it can be read back into core from the overlay area.

Available Storage

Available storage is merely storage currently not in use. Such storage can be assigned as needed.

Memory Links

Memory links are the devices used by the DF MCP to keep track of the organization of core and to note the classifications assigned to core areas. Basically, there are two types of memory links: (1) memory links for in-use storage and (2) memory links for available storage. There is an in-use memory link preceding every area currently being used and an available memory link preceding every unassigned area.

MEMORY LINKS FOR IN-USE STORAGE. An in-use storage link occupies the two words preceding the area it defines. An in-use link provides the following information:

1. It specifies that the area is in-use.

2. It specifies whether the storage is non-overlayable or overlayable.

3. It specifies if the storage contains a program segment or data.

4. It specifies the MIX index of the program using the area. A different MIX index is assigned to each program currently in process by the DF MCP. MIX indexes will be discussed in more detail.

5. It provides the address of the preceding area.

6. It provides the address of the following area. The area referred to as the "preceding area" or "following area" means precisely what it says, no regard is given whether the area is in-use or available.

7. It provides information used to locate descriptors that address the area. This information is needed by the DF MCP if the area is to be overlaid; descriptors addressing the area must be properly marked "not present".

MEMORY LINKS FOR AVAILABLE STORAGE. A memory link for available storage occupies three words, two precede, and one is included in, the area it defines. Such a link provides the following information:

1. It specifies that the area is available.

2. It specifies the size of the area.

3. It provides the address of the preceding area. The area referred to as the "preceding area" or "following area" means precisely what it says, no regard is given whether the area is in-use or available.

4. It provides the address of the following area.

5. It provides the address of the preceding available area.

6. It provides the address of the following available area.

When core storage is classified and organized for the first time after a HALT-LOAD, the DF MCP performs operations to determine which of the eight possible memory modules are available on the system. If any modules are not available, memory links are set up so that the areas in those modules are never assigned and consequently never addressed. Permanent DF MCP segments read in during initialization are marked non-overlayable. Other DF MCP program segments related to initialization routines may be in core after initialization, but they are overlayable. All other core is marked available.

Creating the Available-Disk Table

The Available-Disk Table is created and written on disk at each HALT-LOAD time. To create the Available-Disk Table, the DF MCP first determines the total amount of disk on the system, and then determines the amount of available disk by deducting the areas reserved in the Disk Directory. The Available-Disk Table thus forms the complement of the Disk Directory. The need to create the Available-Disk Table at each HALT-LOAD time follows from the fact that when a permanent file is created on disk it is not entered in the Disk Directory until the program has completed its creation. This procedure must be followed for insurance against events which might cause a program creating a file to be interrupted to the point where the program cannot be recovered (e.g., interrupted by a power failure); thus, leaving file creation at an indeterminable point. Following this procedure, only valid files are entered in the Disk Directory. During the creation of a file, the file's area is reserved because it has been removed from the Available-Disk Table.

Interrogating Peripheral Units

After initialization activities have been completed, control is transferred to the Control Section of the DF MCP. One of the activities of the Control Section is to check for changes in status of I/O units (i.e., check to see if any I/O units changed from READY to NOT READY or NOT READY to READY). This check is made through use of the Interrogate Peripheral Status operator, an operator which places, in the top of the stack, a word representing the current status of peripheral units. One bit in the word is associated with each I/O unit; a unit's bit is 0 if the unit is NOT READY and 1 if it is READY. The word provided by the interrogate peripheral operator is compared with a word that reflects the previously noted status of the units. During initialization, the word used for comparison is set to indicate all units are NOT READY. When a change in status occurs, a request is made to have the STATUS Procedure called.

STATUS PROCEDURE

The STATUS Procedure is responsible, in part, for maintaining the DF MCP tables which contain information, for specifying what units are assigned to programs, and for specifying if units are READY or NOT READY. When an input or input-output unit becomes READY for the first time, STATUS causes the first record on the unit to be read. STATUS then examines pertinent information and enters label information in appropriate tables or specifies that a file is a scratch file, etc. If the first

record on a file is "control card" information, STATUS requests that the CONTROL CARD Procedure be called.

CONTROL CARD PROCEDURE

Program scheduling information, such as instructions to compile a program or to execute a library program, and program parameter information, such as priority specifications, is provided to the DF MCP through use of control cards and program parameter cards. The cards are marked with an invalid character in column 1 and specify their function in word descriptions in a free field format. When control information is read from media other than cards, the means of identification is different, but handling procedures are similar. When the STATUS Procedure calls the CONTROL CARD Procedure, the CONTROL CARD Procedure analyzes the control information and makes appropriate entries in a "schedule sheet". The CONTROL CARD Procedure then requests that the SELECTION Procedure be called.

SELECTION PROCEDURE

The SELECTION Procedure selects a program, if one is available, from the schedule sheet on a priority basis, assigns it a MIX index, and sets up conditions necessary for the program to be initiated.

All programs to be executed must be on the disk as library programs and, therefore, have entries in the Disk Directory. If a program from a library tape is to be run, it may be loaded to the disk through use of a control card. The compilers automatically place programs on disk as library programs; however, for "compile and go" runs, the programs are automatically removed when they are set up to be initiated. SELECTION reads the file header for the file of the program to be initiated. Contained in the file header for a program file is the disk address of the "zero segment" of the file.

The zero segment of a program file is a special segment containing such information as the location within the program file of the program's PRT and Segment Dictionary and the size of each, and the program segment number of the first program segment to be executed. The Segment Dictionary is a table which contains the relative disk address and the size of each program segment in the program. Segments are assigned numbers by the compilers for reference purposes.

The SELECTION Procedure reads the zero segment into core, examines the information, and then reserves areas in core for the PRT (and stack) and the Segment Dictionary according to their specified sizes. These areas are marked non-overlayable. Then the PRT and Segment Dictionary are read into their core areas, and the address of the Segment Dictionary is placed in a reserved PRT cell. From the Segment Dictionary, SELECTION determines the disk address and size of the first program segment and it is read into core. The program can be initiated after the performance of these operations and other necessary "fix-up" operations that may be specified in control information.

To cause a program to be initiated, the SELECTION Procedure constructs an interrupt control word and an interrupt return control word and places them in the program's core stack area. The register settings in these words are given such values that in appearance they indicate that the program was interrupted just before executing its first syllable. SELECTION also places an appropriate initiate control word in the program's PRT and sets up DF MCP tables so that they contain all needed information, including the address of the program's PRT. SELECTION then requests that the RUN Procedure be called and provides the program's MIX index as a parameter to RUN.

RUN PROCEDURE

The RUN Procedure sets up certain variables as needed to initiate a given program. Specifically, one variable assigned a value by RUN is P1MIX. P1MIX is assigned the value of the MIX index which was passed to RUN as a parameter. RUN then transfers control to the INITIATE routine.

MIX INDEX

Programs that are selected from the schedule sheet and put in process are considered to be in the MIX. Every program in the MIX has

been assigned a MIX index. A program's MIX index is actually an index into a DF MCP table called the PRT array. The PRT array contains a descriptor for every program in the MIX. The descriptor for a given program addresses the base of that program's PRT. The descriptor for a particular program is in the PRT array location corresponding to that program's MIX index. Through use of the PRT array and MIX indexes, the DF MCP can locate the PRT of any program in the MIX.

INITIATE ROUTINE

Control is transferred to the INITIATE Routine for the purpose of initiating the program whose MIX index is specified by P1MIX. To initiate the program, INITIATE obtains the Initiate Control Word from the program's PRT which is located through use of the PRT array and P1MIX. Before initiating the program on Processor 1, however, a check is made to see if it could be initiated on Processor 2. If Processor 2 is available and not busy, a variable called P2MIX is given the value of P1MIX and the program is initiated on Processor 2; otherwise, the program is initiated on Processor 1.

P1MIX and P2MIX

After the DF MCP gives control to a normal state program, control will not return to the DF MCP until that program is interrupted. When a normal state program is interrupted, all interrupt information is stored in the program's stack and PRT, and rR and rS are set to control state areas. The DF MCP must, however, have some link back to the program that was processing. P1MIX provides this link for Processor 1 and P2MIX for Processor 2. Before a program is initiated on Processor 1, P1MIX is given the value of the MIX index of the program; likewise, for Processor 2 and P2MIX. Consequently, when an interrupt occurs, the DF MCP knows the MIX index of the program interrupted on Processor 2 and/or Processor 1.

INFORMATION SOURCE FOR A PROGRAM IN PROCESS

As a program continues processing after being initiated, it may soon require additional program segments and/or data which were not provided by the SELECTION Procedure. The principal source of information for a program is its PRT. All simple variables, other than those declared local to procedures, have PRT locations; therefore, they are always present in core while a program is processing. Program segments and data segments are not always present. It was noted above, however, that each program segment and data segment has a descriptor related to it; either a program descriptor (e.g., label descriptor, procedure descriptor) or a data descriptor. These descriptors are located in the PRT. When a program accesses a descriptor, the presence bit of the descriptor will, of course, denote the presence or absence of the information described. If a descriptor is accessed and its presence bit is zero, the presence bit interrupt will be set, control words will be generated and put in place, and subsequently control in Processor 1 will be transferred to the presence bit interrupt location.

PRESENCE BIT ROUTINE

When a presence bit interrupt is detected, control is transferred to the PRESENCE BIT Routine. The fact that a presence bit interrupt occurred means that a program has executed a syllable that caused an attempt to access information described by a descriptor with a zero presence bit. When this situation occurs, the control words for the interrupt contain settings for rC and rL that address the syllable following the syllable that caused the interrupt.

To investigate the interrupt condition, the PRESENCE BIT Routine first locates the PRT of the interrupted program through use of the PRT array and P1MIX. The Initiate Control Word from the PRT is then used to locate the other control words at the top of the program's stack. Through use of the register settings in the control words, PRESENCE BIT locates the syllable that caused the interrupt and, subsequently, the address of the descriptor with a zero presence bit.

To remedy the situation caused by the descriptor with a zero presence bit, the PRESENCE BIT Routine first adjusts the register settings in the control words so they will

reflect the condition that existed before the syllable, that caused the interrupt, was executed. Then the information required by the program is made present.

Methods of Making Information Present

The occurrence of a presence bit interrupt only indicates that a program requires information. The kind of information required is indicated by the descriptor that was accessed. The method in which the information is provided is determined by the kind of information required.

MAKING DATA PRESENT FOR THE FIRST TIME. When a data descriptor with a zero presence bit is accessed for the first time, it contains a code number in bits [33:15] - the address field. The fact that this code is present denotes that no data has yet been related to the descriptor. When this is so, an area in core must be provided and assigned to the descriptor. The size field in the accessed descriptor specifies the kind of storage. (e.g., if the code is 0, overlayable storage will be assigned; if the code is 1, non-overlayable storage will be assigned.) To obtain the required area in core, the PRESENCE BIT Routine calls the GETSPACE Procedure passing parameters that specify: (1) the size of the area required, (2) that the area is for data, and (3) the code indicating if the area is to be overlayable or non-overlayable. GETSPACE provides the area and returns its address to the PRESENCE BIT Routine. PRESENCE BIT then initializes the assigned area with zeros, places the address of the area's descriptor in the area's memory link for use if the area is at some time overlaid, places the address of the area in the data descriptor, and sets the descriptor's presence bit to 1.

MAKING OVERLAID DATA PRESENT. When a data descriptor with a zero presence bit is accessed, its address field may contain an overlay storage address which indicates that the desired information has been overlaid and must be read in from the overlay storage area. As in the above case, PRESENCE BIT must call GETSPACE to obtain an area into which the desired information can be read. Then the address of the area's data descriptor is placed in the area's memory link for use if the area is ever overlaid, the address of the area is placed in the area's descriptor, and the descriptor's presence bit is set to 1. PRESENCE BIT then calls on the DISKIO Procedure which initiates the input operation from the specified address and returns control to the PRESENCE BIT Routine. Then PRESENCE BIT provides that the area used by the data in overlay storage will be returned. At this point, PRESENCE BIT cannot continue until the input operation is complete. Therefore, PRESENCE BIT calls the SLEEP Procedure specifying not to return control until the I/O operation is completed. The control section of the DF MCP then takes control and PRESENCE BIT is temporarily suspended. When control is returned to PRESENCE BIT, the desired information is in the area now addressed by the descriptor.

MAKING PROGRAM SEGMENTS PRESENT. When a program descriptor with a zero presence bit is accessed, the desired program segment must be read in from disk. Program descriptors, however, may address points within a program segment (e.g., a label descriptor addresses a point in a segment to which control may be transferred). Consequently, the address field of a program descriptor with a zero presence bit contains the relative address of the word within the segment that the descriptor must address. To determine a segment's disk address, PRESENCE BIT must examine the segment's entry in the program's Segment Dictionary. Therefore, the program descriptor contains, in addition to the relative address, an index which can be used to locate the segment's entry in the program's Segment Dictionary. Using this index and the address of the Segment Dictionary which was placed in the program's PRT by SELECTION, the Segment Dictionary entry is located and the size and disk address of the segment are obtained. Then, PRESENCE BIT calls GETSPACE to obtain an area into which the segment can be read, places the address of that area in the segment's entry in the Segment Dictionary, and calls DISKIO to initiate the I/O to read the segment into the specified area. Subsequently, the index value for the segment's entry in the Segment Dictionary is placed in the area's memory link, all program descriptors addressing the area are given their absolute addresses (i.e., each descriptor's address field is set to the

value obtained by adding its relative address to the segment's base address), and the presence bits of those descriptors are set to one. PRESENCE BIT then calls the SLEEP Procedure specifying not to return control until the disk I/O is completed. The Control Section then takes control and PRESENCE BIT is temporarily suspended. When control is returned, the desired information is in the area now addressed by the program descriptor(s).

After information has been made present and the concerned descriptor is properly adjusted, PRESENCE BIT transfers control to INITIATE. Then, due to the adjustment made in the program's interrupt words, the program resumes control at the point where it will again attempt to address the desired information, this time successfully.

CONTROL SECTION

The Control Section of the DF MCP performs four principal functions: (1) it interrogates interrupts, (2) it checks for changes in the status of peripheral units, (3) it provides a means by which a DF MCP routine and/or procedure can request that an independent procedure be called, and (4) it provides a means by which a DF MCP procedure can suspend its processing until a necessary condition exists. To perform these functions, the Control Section includes the INDEPENDENT RUNNER Procedure and the SLATE array, the SLEEP Procedure and the BED array, and the NOTHINGTODO routine.

There are two important factors involved with the mechanics of the DF MCP Control Section. One is that a DF MCP Procedure can reserve a core area for use as a private stack. In some cases, a procedure can use the stack area of the normal state program whose interrupt it is handling; in other cases, a procedure must call GETSPACE and obtain a non-overlayable area. In either case, the procedure can set rS to the desired area through use of the STS operator that causes rS to be set to a specified address. The second factor involved with the mechanics of the Control Section is the B 5500 procedure handling techniques; namely, the hardware's generation and use of control words for procedure entry and exit.[9]

[9] Control words and Procedure entry and exit are discussed in Section 1.

Independent Runner Procedure and Slate Array

The INDEPENDENT RUNNER Procedure and the SLATE ARRAY are used when a DF MCP procedure or routine wishes to request that an independent procedure be called. A procedure is termed an "independent procedure" if it is not directly associated with a particular normal state program. For example, the procedures STATUS, CONTROL CARD, SELECTION, and RUN are independent procedures.

To request that an independent Procedure be called, a call is made on the INDEPENDENT RUNNER Procedure specifying the Procedure to be called and a parameter for that Procedure. INDEPENDENT RUNNER then makes an entry in the SLATE array specifying the given information and then returns control to the requesting Procedure.

Sleep Procedure and Bed Array

The SLEEP Procedure and the BED array are used when a DF MCP procedure wishes to suspend its processing until a certain condition exists. An example of when a Procedure must suspend itself is after it has initiated an I/O and it cannot continue until the I/O has been completed. To suspend itself temporarily, a procedure calls the SLEEP Procedure and passes two parameters; one parameter is the address of a word to be tested, and the other is a mask word that specifies which bit(s) in the designated word should be tested. The SLEEP Procedure then makes an entry in the BED array. The information in the BED entry includes: (1) the address of the word specified as the "test word", (2) the mask to be used with the test word, (3) the current value of P1MIX which provides the MIX index of the program that caused the suspended procedure to be called, and (4) the address of the Return Control Word of the SLEEP Procedure. The SLEEP Procedure obtains this value by reading the F-register. After making the entry in the BED array, SLEEP transfers control to the NOTHINGTODO Routine. It is important to note that control is transferred to NOTHINGTODO by branching on a label descriptor. Consequently, the control words generated when SLEEP was called are left in the private stack of the procedure that called SLEEP. It should also be noted that the entry made in the BED contains the address of the Return Control Word of SLEEP.

NOTHINGTODO Routine

The NOTHINGTODO Routine is the principal point of control in the Control Section. When NOTHINGTODO receives control, it first performs an interrogate interrupt operation. Then if there are no interrupts, it checks for an entry in SLATE. If there is an entry in the SLATE and it is then possible to call an independent procedure, NOTHINGTODO sets rS to a stack area for the independent procedure and causes it to be called. If there are no entries in the SLATE, NOTHINGTODO investigates BED. Each time before examining a BED entry, NOTHINGTODO performs an interrogate interrupt operation; then, if there are no interrupts, it selects an entry from the BED. From a BED entry, NOTHINGTODO gets the address of the word to be tested and then obtains the word. The test word is then masked with the mask word provided in the BED entry. A mask is a word containing zero in every bit position other than the positions of bits to be tested. The masking operation is a logical AND operation performed on the test word and the mask. The logical AND operation generates a result word with 1's in the bit positions in which both the test word and the mask have 1's. If the masking operation produced negative results (i.e., the condition required by the suspended program still did not exist), another entry would be tested. If no BED entry provides positive results, NOTHINGTODO performs interrogate interrupt operations and checks for changes in the status of peripheral units. If the masking operation produced positive results (i.e., if the condition required by the suspended program then existed) the NOTHINGTODO routine would remove that BED entry and return control to the suspended procedure.

To return control to a suspended procedure, NOTHINGTODO first sets P1MIX to the MIX index value in the BED entry. Then the F-register is set to the address of the Return Control Word of the SLEEP Procedure. This control word is, of course, still in the private stack of the procedure that suspended itself by calling SLEEP. Then the NOTHINGTODO routine causes an EXIT operation to be performed, just as would be performed to exit a procedure. The EXIT operation is handled by the hardware (see "Exiting a Procedure" in Section 1). Since rF at that time contains the address of the Return Control Word for the SLEEP Procedure, the EXIT operation returns control to the suspended procedure just as though the SLEEP Procedure had caused the EXIT operation.

An example of when a procedure must temporarily suspend itself is after it has initiated an I/O and cannot continue processing until the I/O has been completed. Specifically, consider the case noted above where PRESENCE BIT called DISKIO to initiate a read to obtain information to be made present for a program. In that case, the parameters that PRESENCE BIT would pass to SLEEP would be: (1) the address of the I/O descriptor[10] used to perform the I/O operation, and (2) a mask that specified that the I/O complete bit in the I/O descriptor was to be tested.

The I/O complete bit is the bit that a DF MCP I/O routine sets to 1 when the I/O operation, described by the I/O descriptor, has been successfully completed. With this information and P1MIX and the result obtained by reading rF, SLEEP would make a BED entry and transfer control to NOTHINGTODO. Subsequently, the initiated I/O would be completed by the I/O hardware and an "I/O finished" interrupt would be set. Then the interrupt condition would be detected and control would be transferred to the IOFINISH Procedure. The IOFINISH Procedure would then perform its operations and set the pertinent I/O complete bit to 1. Subsequently, the NOTHINGTODO Routine would take control and test entries in BED. When the entry made for PRESENCE BIT was tested, the test would yield positive results and, through the means noted above, PRESENCE BIT would be reactivated.

GETSPACE PROCEDURE

The GETSPACE Procedure is called by any DF MCP procedure or routine that wishes to obtain an area in core. Parameters needed by GETSPACE specify: (1) the size of the area desired given as number of B 5500 words, (2) the type of information

[10] An I/O descriptor is a special purpose descriptor used to describe an I/O operation.

to be stored in the area (e.g., data or program segment), and (3) the kind of area (either overlayable or non-overlayable). GETSPACE uses one of two algorithms when locating an area to reserve, depending upon the kind of storage requested. After GETSPACE locates an area and reserves it through use of memory links, it "exits", thereby providing the address of the area to the calling routine or procedure.

Locating an Area for Overlayable Storage

If overlayable storage is requested, GETSPACE links through the links for available storage using a special Link List Lookup operator. If a section of available core large enough to fulfill the request is found, that section (or a part of the section if it is larger than necessary) is reserved. To reserve the area, it is removed from the linked list of available storage and marked "in-use" through use of a memory link for in-use storage. If all of the section of available storage were not needed, the part remaining would be linked in with the list of available storage. If no section of available storage is large enough to fulfill the request, GETSPACE first calls the OLAY Procedure to request that a section of overlayable storage be overlayed. Finally, through use of overlay, a large enough section of available storage is obtained and reserved.

Locating an Area for Non-Overlayable Storage

If non-overlayable storage is requested, GETSPACE always obtains an area as near as possible to the lowest addressed portion of memory. This is done in an attempt to keep all non-overlayable storage together in order to reduce the possibility of such storage causing overlayable storage to be checkerboarded. To obtain non-overlayable storage, GETSPACE begins by examining the first memory link following the non-overlayable portion of the DF MCP. If this link specifies that the defined storage is available, GETSPACE examines its size. If it is of sufficient size, it is reserved; otherwise, GETSPACE examines the next area. It should be noted that there never exist two adjacent areas of available storage. When a situation occurs where this would happen, the two areas are joined. If a link specifies that an in-use area is overlayable, then OLAY is called to overlay the area. If the overlaid area or the overlaid area added to an adjacent available area, whichever the case may be, is of sufficient size, it is reserved. Otherwise, this process continues until an area of sufficient size is provided.

OLAY PROCEDURE

It is the function of the OLAY Procedure to overlay in-use areas in core in order to change them to available areas. However, before overlaying an area, OLAY must know what program has been assigned the area and what the area contains (i.e., a program segment or data). This information, as was noted in the paragraph on classification of disk storage, can be determined by examining the memory link for the area. First of all, the information must be used to determine whether or not the area can be overlaid; then, if the area can be overlaid, the information is needed to determine the algorithm to use when performing the overlay operations. The OLAY Procedure considers in-use storage to fall into one of three categories: (1) data storage, (2) program segment storage for object programs, or (3) DF MCP storage. DF MCP storage is recognized because the DF MCP uses MIX index zero. Program segment storage is distinguished from data storage through use of the type-code kept in the memory link.

Overlaying a Data Area

When an area to be overlaid contains data, OLAY examines the Initiate Control Word from the PRT of the program assigned the area. The location of the pertinent program's PRT is determined through use of the MIX index obtained from the memory link. From the Initiate Control Word, OLAY can determine if the program using the area was interrupted during word mode operation or character mode operation.

If the program using the area was interrupted during word mode operation, OLAY immediately calls the procedure DISKIO to initiate an I/O operation which will place the data in overlay storage. Using the address placed in the area's memory link by PRESENCE

BIT, OLAY locates the area's data descriptor. The presence bit of that descriptor is set to zero and the descriptor's address field is set to the address of the data in overlay storage. Also, since it is possible that the program's stack may contain descriptors that address the overlaid area (e.g., parameters to procedures), OLAY searches the stack, locates, and properly marks all such descriptors absent. OLAY then calls the SLEEP procedure passing: (1) the address of the I/O descriptor used to initiate the write to overlay storage, and (2) a mask for the I/O complete bit. When the I/O is completed, OLAY is re-initiated and calls FORGETSPACE which will mark the area available.

If the program using the area to be overlaid was interrupted during character mode, OLAY would overlay the area only if no other overlayable area was available. This is so because a character mode procedure can store absolute address as operands in its stack. This facility makes it necessary for OLAY to not only search a program's stack for descriptors containing an address within the area to be overlaid, but also to check the fifteen low order bit operands for such addresses. This added necessity has two disadvantages: (1) it is time consuming compared to checking only for descriptors (a stack search for descriptors is made using the Flag Bit Search operator designed to be especially fast and efficient for such use), and (2) if the fifteen low order bits of an operand in the program's stack appear to address the pertinent area, the area cannot be overlaid.

Overlaying Program Segment Areas

When an area to be overlaid contains a program segment of an object program, OLAY obtains the rC setting stored in the Interrupt Return Control Word of the program using the area. (The address of the interrupt Return Control Word is obtained from the Initiate Control Word in the program's PRT.) OLAY then checks to see if the rC value (i.e., the address to which control will be transferred after interrupt handling) is the address of a word within the area to be overlaid. If so, the area is not overlaid. If the rC address is not within the address range of the segment to be overlaid, OLAY obtains the address of the program's Segment Dictionary from the program's PRT. Using the Segment Dictionary index placed in the area's memory link by PRESENCE BIT, OLAY locates the Segment Dictionary entry for the segment to be overlaid and thus obtains the segment's base core address. Subsequently, OLAY proceeds to locate all program descriptors in the program's PRT and stack that contain addresses within the address range of the segment to be overlaid. When such a descriptor is found, OLAY calculates the relative address of the word within the program segment that the descriptor addresses. This is obtained by subtracting the segment's base address from the address in the descriptor's address field. This relative address is then stored in the descriptor's address field for further use by PRESENCE BIT, if needed, and the descriptor is marked absent. OLAY then calls FORGETSPACE to mark the area available.

Overlaying a DF MCP Segment

DF MCP segments cannot be overlaid in the same fashion as object program information. This is so because DF MCP segments are always referenced by the DF MCP in control state. As was noted in Section 1, processing in control state is never interrupted and interrupts such as presence bits are inhibited. Since descriptors with zero presence bits are not detected if accessed during control state operation, the presence bit is not used to indicate that DF MCP segments are absent. Instead, if a DF MCP segment is absent from core, the address field in its descriptor is provided with the address of the procedure ESPBIT — a non-overlayable DF MCP procedure. This, of course, means that if an attempt is made to transfer control to a DF MCP segment that is absent from core, control will instead be transferred to ESPBIT.

Because of the special handling of descriptors for DF MCP segments, OLAY needs only to place the address of the ESPBIT Procedure in such a descriptor to mark it absent. However, before overlaying an area containing a DF MCP segment, OLAY must ensure that the segment is not in core due to a current need of the DF MCP. For example, OLAY would not want to overlay the PRESENCE BIT procedure which had itself called OLAY and which would receive control when

OLAY exited. Also, OLAY would not want to overlay a procedure that had been temporarily suspended due to a current need of the DF MCP. For example, OLAY would not want to overlay the PRESENCE BIT procedure which had itself called OLAY and which would receive control when OLAY exited. Also, OLAY would not want to overlay a procedure that had been temporarily suspended due to a call on SLEEP or any procedure that had called a procedure that has been temporarily suspended, etc. Consequently, before OLAY overlays a DF MCP segment, it first searches its own stack for Return Control Words containing a rC value addressing a word within the segment to be overlaid; if any are found, the segment is not overlaid. Also, OLAY investigates stacks related to entries in the BED. To do this, OLAY obtains, from the BED entry, the address used by NOTHINGTODO to return control to a suspended procedure. This address, of course, points to the stack of the suspended procedure and provides OLAY with means to locate the Return Control Words in that stack. Therefore, OLAY can determine if the segment to be overlaid would be needed when a suspended procedure were reactivated and, if so, the segment would not be overlaid. If OLAY, after making all necessary tests, determines the DF MCP segment can be overlaid, the segment's descriptor is provided with the address of ESPBIT, and FORGETSPACE is called to mark the area available.

FORGETSPACE PROCEDURE

The function of the FORGETSPACE Procedure is to change the classification of an area from "in-use" to "available". When FORGETSPACE is called, it is provided with the address of an in-use area which is to be marked available. However, before marking an area available, FORGETSPACE (through use of memory links) determines the availability of the areas adjacent.

First, the following adjacent area is checked. If it is not available, it is left as is. If it is available, its memory link is removed from the memory lists and it is made part of the area defined by the link for the previously in-use area. In either case, the preceding adjacent area is then checked.

If the preceding area is not available, the area defined by the link for the previously in-use area is entered in the memory lists as available and FORGETSPACE is finished. If the preceding area is available, the link for the previously in-use area is removed from the memory list. Then the area defined by the link for the previously in-use is made part of the area defined by the link for preceding adjacent area and FORGETSPACE is finished.

ESPBIT PROCEDURE

The ESPBIT Procedure is responsible for bringing DF MCP segments into core. This procedure, rather than PRESENCE BIT, must be used for DF MCP segments because the presence bit interrupt feature is not available while operating in control state. The performance of ESPBIT is centered around the set-up of procedure descriptors for overlayable DF MCP segments. When the DF MCP is compiled, the ESPOL compiler determines the disk addresses for all segments and the core addresses for all non-overlayable segments. This information is used when setting up entries in the PRT for the DF MCP.

When procedure descriptors for overlayable DF MCP segments are constructed by ESPOL and placed in the DF MCP's PRT, the field at [18:15] in these descriptors is set to the disk address of their respective segments. Also, the size field in a descriptor is set to the segment's size. However, the address fields of procedure descriptors for overlayable segments are set to the core address of the first non-overlayable segment of the DF MCP. By design, the first such segment is that of the procedure ESPBIT. When the DF MCP is executing and the descriptor for a non-present segment is addressed, the set-up of the segment's procedure descriptor causes control to be transferred to ESPBIT. It is then the function of ESPBIT to provide that control be given to the procedure on which the call was intended.

In order to determine what procedure should have taken control, ESPBIT obtains the rC and rL values from the Return Control Word that was placed in the stack when the procedure was called. Using these values, the syllable that caused the procedure call is located and, subsequently, the procedure

descriptor which was accessed is located. This, of course, is the procedure descriptor for the desired segment. Using the information in the descriptor, ESPBIT then calls GETSPACE to reserve an area and subsequently reads the segment into core. ESPIT then places the PRT address of the procedure descriptor in the area's memory link for use by OLAY, and places the address of the area in the procedure descriptor. Then, by transferring control to the segment, the desired procedure assumes control under the same conditions as would have existed if ESPBIT had not intervened.

OBJECT PROGRAM I/O FACILITIES

The handling of object program I/O facilities on the B 5500 is a function of the DF MCP as well as the object program. It is the responsibility of the object program to specify, for each file used, the file handling techniques such as: (1) number of buffer areas to use, (2) size of buffer areas, (3) blocking techniques, and (4) in the case of disk, the record accessing technique (i.e., serial, random, update). Also, for each I/O statement, the object program must specify the file and I/O action to be used, and the control data movement to and from buffer areas. However, it is the responsibility of the DF MCP to locate files, to provide buffer areas and handle their use, to perform blocking and record accessing, and to execute I/O operations to read or write the files.

I/O Intrinsics

In the area of I/O, the DF MCP provides a number of procedures that execute in normal state rather than control state. Such procedures are called "intrinsics". When a compiler determines that a program requires the use of an intrinsic, a procedure descriptor for the intrinsic is placed in the program's PRT. The program can then make calls on the intrinsic in the same fashion as any other procedure.

Specification of File Handling Techniques

File handling techniques for object program files are specified in the source language representation of the program. In a source program, before the file identifiers are used in I/O statements, each file identifier is associated with the file handling techniques to be used with the file. In this section of the program, the file identifier is also associated with the file name of the file concerned. However, at run time, it is possible to associate a file identifier with a different file name.

FILE AND FILE NAMES. In respect to file names, there are two types of files on the B 5500; standard files and non-standard files. A standard file is a file which has a file name physically associated with it. A non-standard is a file that requires outside intervention to have a name associated with it. In the case of files on disk, all files are standard. Names are associated with these files through the disk directory, as was noted in Section 2. Names are associated with other standard files by "standard labels". A standard label is a record with a given format that appears as the first record in a file. One of the entries in a standard label is the file's name. Files that do not have standard labels are non-standard files. An example of a non-standard file would be a magnetic tape without a standard label. To associate a file name with a non-standard file requires that special information be provided to associate the file name with the I/O unit where the file is located. The information may be supplied through use of a "label equation card" or an operator message.

FILE PARAMETER BLOCK. Each program for the B 5500 has a File Parameter Block (FPB). The FPB is created when a program is compiled, and later modified by the SELECTION routine during the "fix-up" before a program is initiated. The FPB for a program has an entry for every file to be used by the program.

When a file is declared in a program, that is, when the source program associates the file identifier with a file name and file handling techniques, the compiler assigns the file identifier a file number. This file number, rather than the file identifier, is then used in all references made to the corresponding file by the object program. For each file number, and in file number order, there is an entry in the program's FPB. Each entry in the FPB contains the file identifier,

the multiple file identification, and the file identification for the particular file number. The location and size of the FPB are placed in an entry of the program's zero segment. When the SELECTION Procedure is performing "fix-up" operations, it uses this information to obtain the FPB. The FPB must be used at this time to process label equation cards, if any.

Label equation cards are special program parameter cards that can be used at run time to associate a file name with a file identifier used, in the source language representation of a program. Each label equation card contains the file identifier concerned, and the equation information which includes the multiple file identification and file identification to be associated with the file identifier. When SELECTION obtains a program's FPB, it also obtains all label equation cards for the program, if any. Then the file identifiers in the FPB entries are compared with the file identifiers on label equation cards. If a match is found, information in the FPB is replaced with the corresponding information from the label equation card. It is in this way that file names, associated with files represented by file identifiers, can be decided at run time. After all label equation cards for a program have been handled, SELECTION modifies the FPB again by removing the file identifier entries, which are no longer required. Then a descriptor containing the address of the compacted FPB is placed in a specified location in the object program's PRT. Using this descriptor and a file number, the object program is able to make all necessary references to FPB entries.

FILE INFORMATION BLOCKS. At run time, there is one File Information Block (FIB) generated for each file to be used by a program. An FIB is generated by an object program at each program point corresponding to a file declaration in the source language representation of the program. Initially, the FIB contains only the information about file handling techniques provided in the source program. When a file is put to use, I/O routines use a file's FIB to store information pertinent to the file such as block counts, record counts, etc. At the point when a file's FIB is created, a buffer descriptor area, containing an I/O descriptor for each buffer area to be used for the file, is also created.

LOGICAL UNIT NUMBERS. As was noted in the section about the STATUS Procedure, the Burroughs B 5500 has a special syllable which, when executed, provides a result giving the status of every peripheral unit. The bits in this result word are assumed to be numbered from right to left, starting with 0. The number of the bit representing a particular I/O unit is taken to be that unit's "logical Unit Number". References to I/O units made by the I/O Procedures and Routines of the DF MCP are made through use of the unit's logical number.

FILE NAMES VS. I/O UNITS. The association of file names with files on disk through use of the disk directory is straightforward since disk file locations are relatively static. The association of file names with I/O units other than disk, however, requires checking each time STATUS notes that a unit not in use is made READY. Since files on units such as magnetic tape units are provided by a system operator, the DF MCP must have a dynamic directory for keeping a record of what files, if any, are on what units. This directory is, in fact, made up of three tables: the LABEL Table, the MULTI Table, and the RDC Table. There is an entry in each of these tables for each unit that may be on the system. Entries within each table are kept in Logical Unit Number order.

The LABEL Table is the primary table in the group. A unit's entry in this table specifies one of the following: (1) the unit is NOT READY, (2) the unit is READY and contains a file that can be used for output (e.g., a line printer file, or a magnetic tape file with a write-ring), (3) the unit is READY and contains an input file not in-use (the LABEL TABLE entry in this case would include the file identification of the input file), or (4) the file is READY but in-use.

The MULTI Table contains the multiple file identification of the file, if any, on the unit represented by the table entry. The RDC Table contains the reel number, purge date, and cycle number of the file, if any, on the

unit represented by the table entry. Information in the LABEL Table, the MULTI Table, and the RDC Table is obtained from the standard labels on the files, if the files are so labeled; otherwise, the information can be supplied through use of Label Equation Cards or operator messages. The STATUS Procedure has the primary responsibility of maintaining these tables.

Opening a File

When a program first requires a file, the file must be "opened". The process of opening a file includes locating the file and providing buffer areas. When a file is to be opened, a DF MCP FILE OPEN Procedure is called. If a program requires an input file, the FILE OPEN Procedure locates the program's FPB to obtain the name of the file. If the file is a disk file, the Disk Directory is searched and the file header for the desired file is read into core. Otherwise, the LABEL Table, the MULTI Table, and RDC Table are searched to locate the unit containing the file and mark it in-use. If a file cannot be located, a message is typed to notify the operator. Then buffer areas, as specified in the file's FIB, are obtained by calling GETSPACE. Subsequently, the I/O descriptors in the file's buffer descriptor area are provided with the addresses of the buffer areas, and the buffer areas are filled with records from the input file.

If a program required an output file, the type of file must be specified (i.e., magnetic tape, disk, etc.). Buffer areas and descriptors for output files are obtained as noted above; then, if the file is for disk, a disk area of the size specified will be obtained from the areas noted in the Available-Disk Table. If another type file is desired, the LABEL Table is searched and, if a file is available, it is assigned; otherwise, a message is typed to notify the operator that the file is needed. If a line printer is requested and not available, a check is made to see if the program specified a printer back-up option. If not, a message is typed to notify the operator that a printer is required, and SLEEP is called to await a printer. If the back-up option is specified, the FILE OPEN Procedure sets the "continuity bit" in the file's I/O descriptors. The significance of setting this bit will be noted later.

BUFFER AREA ACCESSED BY OBJECT PROGRAMS. As was noted, there is a buffer descriptor area for each file to be used by a program. The buffer descriptor area may be set up to contain one or more I/O descriptors, as specified by the program. And, because of the technique used to handle these descriptors, a program is not programmatically dependent upon the number of buffer areas assigned. In general, I/O descriptors in buffer descriptor areas are handled in the following manner. The program always accesses the top I/O descriptor in the area. When the program has completed its use with the area addressed by the top descriptor, the area is "released". Then the DF MCP performs an I/O to refill or write-out the buffer, whichever is the case. Also, the DF MCP "rotates" the I/O descriptors; that is, if the buffer area is set up for more than one I/O descriptor, the descriptors below the top descriptor are moved up and the previous top descriptor is moved to the bottom of the area. Then control is returned to the object program which can continue accessing the area addressed by the "top descriptor".

It should be noted that each time before accessing the top I/O descriptor, the intrinsics that handle I/O will check the "I/O Finish" bit in the descriptor. This is done since it would be possible for an I/O descriptor to come to the top before the I/O on its buffer area was complete. If the program finds the I/O for the buffer has not been completed, a "SLEEP Communicate" is performed, passing the address of the I/O descriptor and a mask for the I/O Finish bit.

The SLEEP Communicate is performed by placing SLEEP Procedure parameters in the program stack together with a code specifying the type of communicate, and then executing a communicate operator. The DF MCP subsequently makes a call on the SLEEP Procedure. Then, when the necessary condition occurs, control is transferred back to the program and it can continue processing.

Communicate

The general way in which object programs interact with the DF MCP is implicitly through interrupt action such as with presence bit interrupts. Also, as noted previously,

the intrinsic technique is used so that an object program and DF MCP can interact without a need for special entries to control state. In other cases, however, a special operator (the communicate operator) is used so that control can be explicitly transferred from normal state operation to control state operation.

To use the communicate operator, a normal state program first places parameters in its stack; then the communicate operator is executed. The Communicate operator causes a "communicate interrupt". The DF MCP routine that handles this interrupt first locates the stack of the program that caused the interrupt. Then, according to a code value in the parameters in the program's stack, the DF MCP transfers control to the section of the DF MCP designed to handle a communicate interrupt with that code.

Performance of Object Program I/O by the DF MCP

To release a buffer area that is to be refilled or written-out, a normal state program performs a "program release" operation. This operation is performed by first placing, in rA, the address of the I/O descriptor that addresses the area to be released; then a Program release syllable is executed. The Program Release syllable causes the address of the I/O descriptor to be placed in the program's PRT at rR + 9 and checks the continuity bit in the I/O descriptor. If the continuity bit is set, the continuity bit interrupt is set; if not, the Program Release interrupt is set. The setting of the interrupt causes the program to be interrupted and, subsequently, control is transferred to the interrupt location for the interrupt. From this point, the PROGRAM RELEASE Procedure or CONTINUITY BIT Routine is called.

PROGRAM RELEASE PROCEDURE. The PROGRAM RELEASE Procedure obtains the address of the I/O descriptor from the program's PRT and sets the presence bit of the descriptor to zero. (The presence bit of an "in-process" I/O descriptor is set to zero so that a presence bit interrupt will occur if the descriptor is accessed before the I/O is complete. Since I/O intrinsics check an I/O descriptor before accessing it, this is not likely to happen. However, when using Stream Procedures and RELEASE statements in Extended ALGOL, it is possible for a program to make such an access. The PRESENCE BIT Routine remedies the situation, should it occur.) Then the IOREQUEST Procedure is called. When IOREQUEST returns control, the I/O descriptors in the buffer descriptor area are rotated, as discussed above, and finally, control is transferred to the INITIATE routine.

CONTINUITY BIT ROUTINE AND PRINTER BACKUP PROCEDURE. The CONTINUITY BIT Routine checks to see if the I/O descriptor that caused the interrupt was a line printer descriptor, as would be the case for printer backup files. If this is the case, the procedure that handles printer backup is called. The PRINTER BACKUP Procedure then writes the print line, together with a copy of the I/O descriptor which would have been used to write the line on a printer backup file. The remainder of the handling of the I/O is as described for the PROGRAM RELEASE Procedure. At a later point in time, when a line printer is available, the printer backup file can be printed. The I/O descriptor that accompanies a line of print on the backup file is used when the line is printed. Consequently, the format of the printout will be as designated by the program that created it.

IOREQUEST PROCEDURE. It is the function of the IOREQUEST Procedure to either initiate an I/O, or queue up an I/O descriptor in the I/O Queue. The I/O Queue is made up of four tables; the IOQUE, the FINALQUE, the LOCATQUE, and the UNIT table. When a request is made to perform an I/O, the request is queued with respect to logical unit number. Each request for an I/O requires an entry in IOQUE, FINALQUE, and LOCATQUE. IOQUE contains a copy of the I/O descriptor for the request. FINALQUE contains information about a descriptor to be returned to the normal state program after the I/O is completed. LOCATQUE contains the address of the top I/O descriptor in the pertinent buffer descriptor area, the MIX index of the program that made the request, the logical unit number of the I/O unit, and an index to the next request queued for the unit. The UNIT table contains information about each I/O unit. Entries in the UNIT table are in logical unit number order. An

entry for a particular unit includes information specifying the type of unit (e.g., magnetic tape, card reader, etc.), if the unit is READY, if the unit is currently processing an I/O, and if the unit is awaiting error recovery.

When IOREQUEST is called, a check is made to see if the I/O Queue is full. If it is, the SLEEP Procedure is called and IOREQUEST will not continue until space in the queue is available. When queue space is available, a check is made to see if any other I/O's for the unit are queued or in process. If so, the current request is linked into the list of requests for the unit. If not, a test is made to see if an I/O channel (i.e., I/O control unit) is available. If a channel is available, the INITIATEIO Procedure is called to initiate the I/O. If a channel is not available, the QUEUEUP Procedure is called to enter the logical unit number of the unit in WAITQUE, a table used by the IOFINISH Procedure.

INITIATEIO PROCEDURE. The INITIATEIO Procedure is called when a unit is ready for an I/O and an I/O channel is available. INITIATEIO makes use of a table called CHANNEL which has an entry for each I/O Channel. In the entry for the channel to be used for the I/O, INITIATE stores the logical unit number of the unit which will perform the I/O. Then the I/O is initiated and note is made of I/O timing information for logging.

IOFINISH PROCEDURE. When an IOFINISH Interrupt occurs, the IOFINISH Procedure is called. At this time, at least one I/O channel is available. IOFINISH first checks to see if the result descriptor, for the completed I/O, noted any errors. If so, action is taken to prohibit further I/O on the unit until the error condition is rectified; however, error conditions, if any, are not handled at this time. Instead, a check is made to see if any units in WAITQUE are waiting for an I/O channel. If so, an I/O is initiated on the first unit noted in WAITQUE. If there are no entries in WAITQUE, a check is made to see if another I/O can be made on the unit that just completed an I/O. If so, an I/O is initiated for that unit. If an I/O was initiated on a unit from WAITQUE, and a second unit which just completed an I/O was ready for another I/O, the second unit would be queued in WAITQUE.

After all actions to initiate I/O's for queued requests have been completed, IOFINISH proceeds to have error conditions rectified, if any, or performs final handling of the I/O request. If errors were noted, IOFINISH would call INDEPENDENT RUNNER to request that the IOERR Procedure be called. IOERR would then take action to rectify the error. If no errors occurred, IOFINISH would perform final handling of the request, including setting the I/O finish and presence bits in the I/O descriptor to 1. After completing all adjustments, the buffer area concerned is ready for further program use.

A NOTE ON PARALLEL PROCESSING

As was noted in the paragraph describing the INITIATE Routine, programs are initiated on Processor 2 in favor of Processor 1. This is done because Processor 1 handles control state operations, and processor 2 can only idle if no normal state program is initiated on it. To obtain maximum use of Processor 2, interrupts must be handled in such a way that Processor 2 need never idle if a normal state program is ready to be initiated. Consequently, interrupts are handled on a priority basis. That is, each interrupt has a priority and those of higher priority are handled first. Processor 1 interrupts have a higher priority than Processor 2 interrupts. Consequently, if both a Processor 1 interrupt and a Processor 2 interrupt were set at the same time, the Processor 1 interrupt would be handled by the time the Processor 2 interrupt was interrogated.

When a program on Processor 1 generates an interrupt condition, it may be such that the condition cannot be immediately rectified. For example, when a presence bit interrupt occurs, an I/O operation to bring in program segment may have to take place. Consequently, that program identified by P1MIX is temporarily "suspended" by calling SLEEP. When the program identified by P1MIX is suspended, NOTHINGTODO is called and P1MIX is set to zero. Subsequently, NOTHINGTODO re-initiates a procedure from the BED, and P1MIX is assigned the value of the pertinent program's MIX index. It should be recalled that processing on Processor 1 can also be interrupted by the occurrence of

an interrupt related to Processor 2. Therefore, it is often the case that the program on Processor 1 is only interrupted so that the DF MCP can handle a Processor 2 interrupt. When this is the case, however, P1MIX will not have been set to zero; it will still contain the value of the MIX index of the program from Processor 1. It should also be noted that when this is the case, the program identified by P1MIX is ready to be initiated; it requires no DF MCP "fix-up". Because of the noted conditions, Processor 2 interrupts are always handled as follows:

1. The values of P1MIX and P2MIX are exchanged one for the other.

2. Then, if P2MIX is not zero, the program whose MIX index is specified by P2MIX is initiated on Processor 2 and operation would continue as noted by step 3 below. If P2MIX is zero, operation would immediately continue as noted by step 3 below.

3. Control is transferred to the Processor 1 interrupt location that corresponds with the pertinent Processor 2 interrupt, and the interrupt is handled.

It should be noted that interrupt conditions are not related to processors, but rather to programs. When a program is interrupted, all information required for its re-initiation is contained in control words stored in core. Consequently, it makes no difference if values of P1MIX and P2MIX get exchanged or that all interrupts become associated to P1MIX.

BREAKOUT, RESTART, AND EMERGENCY INTERRUPT FACILITIES

The DF MCP provides facilities that allow programs to have rerun points and also allow operator initiated emergency interrupts. If a program requests a breakout, or if an operator requests an emergency interrupt, all processing of object programs is halted. Subsequently, all of memory and overlay storage is written on magnetic tape; then, in the case of a breakout, object programs are re-initiated and continue processing. In an emergency interrupt situation, processing is terminated so that the system is free for other use.

When a program is to be restarted at a rerun point, or programs interrupted by an emergency interrupt are to be re-initiated, no programs may be on the system. Also, all files related to the program(s) to be restarted must be in place on the units where they were at breakout or emergency interrupt time. At such a time, a restart request will be handled by reading the restart information and restoring core to the condition that existed when the breakout occurred. Then overlay storage is restored. Finally, if an emergency interrupt is being restarted, all programs are set up to be re-initiated. If a program is being restarted from a breakout point, only that program is restarted. Other programs, which may have been in process when the breakout occurred and which are reflected in the restored memory and overlay storage, are terminated. The primary procedure used to perform breakouts, restarts, and emergency interrupts, is the BREAKSTART Procedure.

Breakout and Emergency Interrupt

When a Breakout or Emergency interrupt is requested, the BREAKSTART Procedure is called, with a parameter specifying either a Breakout or Emergency interrupt. BREAKSTART first halts the program on Processor 2 (if any), takes action to inhibit any programs to be re-initiated by NOTHINGTODO, and then adjusts the I/O Queue so that no object program's I/O requests (if any) appear to be queued up. All overlayable core is then overlaid. BREAKSTART then obtains a buffer area to use when writing disk and an area into which the code, from interrupt location areas, can be placed. Then, in the case of a breakout, a check is made to see if a breakout tape, for the subject program, has been assigned previously. If so, its location is noted; if not, or in the case of an emergency interrupt, a tape is acquired. Then BREAKSTART moves the code from interrupt locations that could be branched to during the time that core is being written. The code placed in these interrupt locations will handle interrupts as required by the REDUMP Procedure. The REDUMP Procedure will control operations for writing

memory. Then REDUMP is called, with an I/O descriptor containing a zero address. (The I/O bit in this descriptor is set for writing.)

REDUMP PROCEDURE. REDUMP is a non-overlayable DF MCP procedure. When REDUMP is called, it first checks the I/O bit to determine if it was called for either a breakout or restart. When called for a breakout or emergency interrupt, REDUMP obtains the Return Control Word from its stack and places it in a reserved cell in the DF MCP's PRT. This will be needed at restart time. The REDUMP begins dumping memory, 512 words per write. It does not return control to BREAKSTART until all of memory is dumped, unless a tape writer error occurs. If an error occurs, BREAKSTART provides that the error condition is remedied, and REDUMP continues dumping memory. Control is finally returned to BREAKSTART which checks to see if a breakout or restart is being performed. Finding it is a breakout, the interrupt locations are restored to their original settings, and the DF MCP I/O Procedures are used to write the overlay storage on tape. Then, after all restart information is written on tape, the programs previously in process are terminated if an emergency interrupt was requested. If a breakout was requested, the I/O Queue is readjusted so the queued I/O's, if any, can be processed and the other programs are allowed to be re-initiated.

Restart

When programs are to be re-initiated at rerun points, or emergency interrupt points, no programs may be on the system. Also, the system configuration must be identical to the one that existed when the breakout or interrupt point being re-established occurred, and files for the program(s) being restarted must be in place.

When a restart is requested, the BREAKSTART Procedure is called, with a parameter specifying that a restart from a rerun point, or emergency interrupt point, is to be made. Also, information specifying where the restart information is located must be provided. Then a buffer area for "reading memory" is obtained in the high address section of memory and the restart tape is positioned. Then the interrupt locations that may be branched to, while reading memory, are set to the code required by BREAKSTART. (This handling of interrupt locations is the same as when memory was to be written, except the original locations need not be saved this time.) At this time, REDUMP is called and provided with an I/O descriptor set to read into the buffer area provided.

REDUMP checks the I/O bit in the I/O descriptor, and finding a restart is being performed, reads the first 512 words from the restart tape. This information is read into the buffer area provided, and then moved to the first 512 words of core. Then REDUMP moves the stack area, currently used by itself and BREAKSTART, to an area starting at location 100. (This is within the area reserved as the initial stack for interrupt handling routines of the DF MCP, and, consequently, need not remain as it was at the breakout point.) Then rS is set so that REDUMP will use this new area. (This area must be used because, when memory is read, the area previously used by REDUMP for a stack will be overwritten. Since REDUMP is a non-overlayable Procedure, its location is constant.) After relocating its stack, REDUMP obtains the Return Control Word from the DF MCP's PRT which was just read in from the restart tape. This Return Control Word is placed in REDUMP's stack so that when REDUMP exits, control will be returned to the copy of BREAKSTART that will be read in from the restart tape. Then REDUMP reads the remainder of the core information directly into the memory locations from which it originated. Subsequently, control is returned to the copy of BREAKSTART that is currently in core.

BREAKSTART then checks to see if this is a breakout or restart. Then, finding it is a restart, the interrupt locations are restored to their initial code and the DF MCP I/O Procedures are used to read and restore overlay storage.

Finally, if this is a restart at a rerun point, programs, other than the one to be restarted (if any), are terminated; otherwise, they are

left as is. Then files are positioned to be in accord with the conditions at breakout time, the I/O Queue is readjusted to allow I/O, and the program(s) is allowed to be re-initiated.

It should be noted that when a breakout is performed, no action is taken to record information in user files (including files on disk) in the restart information. Consequently, if files are in any way altered after a breakout, the restart information will not reflect the changed conditions of the files.

CHARACTERISTICS OF THE PROGRAM LOGGING FACILITY

The DF MCP keeps a log of processor times and I/O times used for every program run on the system. The log is kept in a file on the user portion of the disk. The user is required to provide the file.

Information is placed in the log by SIGNOFF, a DF MCP procedure. SIGNOFF obtains the information from DF MCP tables, following the completion of a program. If the program being logged is an object program that was called from library to execute, or a compiler that was called for syntax checking only, or a compiler that was called to compile to library, or a compiler that compiled a program with syntax errors, the log information is written in the log immediately following the completion of the program. However, when a "compile and go" run is called for, the compiler's log information is written on system disk, immediately following compilation. Then after running the compiled program, the log information for the compiler is read from system disk and written in the log. Then the object program's log information is written immediately following the log information from system disk.

A log entry for a compiler or object program contains the following:

1. Control Card Information. The first seventy-two columns of information from the control card used to have the object program or compiler called for execution is provided.

2. Today's Date. This entry specifies the current date. (The date is provided each day by the system operator. The value for this date is retained until changed by the operator.)

3. Start Time. This entry specifies the time of day when the program was initiated. The hour of day within this entry is based on a 24 hour day. (The initial time of day is provided by the system operator. It is continually updated by the DF MCP, through use of the B 5500 timer. The timer records time changes each 1/60th of a second and causes a timer interrupt every 64/60th of a second.)

4. Finish Time. This entry specifies the time of day when the program was completed.

5. I/O Channel Time. This entry specifies the amount of time the program required use of I/O channels (i.e., I/O control units).

6. Process Time. This entry specifies the amount of time directly related to the processing of the program.

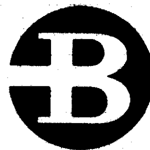
7. Pro-rated Time. This entry specifies the amount of time required by the program for functions not directly related to the processing of the program. Specifically, it reflects the amount of time that a program is suspended from processing (i.e., entered in the BED), awaiting a required condition. For example, a majority of this time can be attributed (in the case of "I/O bound" programs) to object program I/O. Also, this time can be related to I/O time required for overlay operations; therefore, it should be noted that pro-rated time may not be identical, from one running of a program to another. That is, the amount of pro-rated time assigned to a program is dependent

upon factors such as: the number of programs in the MIX, storage requirements of programs in the MIX, system configuration (i.e., number of memory modules, processors, and I/O channels), data requirements, etc.

8. File Information. There is a record in the program's log entry for each file used by the program. The length of time the file was open and the unit used by the file are specified.

The process time and pro-rated time listed for programs represent at least 98% of the acutal processor time used by the program. Information in the log is not lost due to HALT-LOAD operations. When the log becomes half full, a message is typed to notify the operator. A message is also printed when the log is full. When the log is full and space is required for another entry, SIGNOFF assumes the log to be empty and enters the information as the first entry in the file.

A program to print the log is provided; however, any program can read the log file. Consequently, each installation can provide its own log printing program, and format the output as desired.



Wherever There's
Business There's **Burroughs**

1023579 10-66 Printed in U.S. America