SOFTWARE DEVELOPMENT LANGUAGE

FOR THE

BURROUGHS B1700

by Donald R. McCrea

Burroughs Corporation
Santa Barbara Plant
Goleta, California

ABSTRACT

The Burroughs B1700 is a small, general-purpose computer which is (a) dynamically microprogrammable, and (b) designed to support hundreds of independent, special-purpose machine architectures. Each language that runs on the B1700 has its own interpreter. In keeping with this flexibility, a language and underlying machine structure were designed to be used for implementation of the operating system and for implementation of the different compilers. This is an ALGOL-like, GO TO-free language which is elegant, and yet modest in its design. The underlying machine structure is highly stack-oriented, allowing re-entrancy, recursion, and up-level addressing.

Keywords: software implementation language, procedure-oriented language, stack machine, microprogramming, computer architecture, B1700, S-language, interpretation, GO TO-free

SOFTWARE DEVELOPMENT LANGUAGE

FOR THE

BURROUGHS B1700

by

Donald R. McCrea

Burroughs Corporation

SOFTWARE DEVELOPMENT LANGUAGE

FOR THE

BURROUGHS B1700

I. Introduction

The Burroughs B1700 is a small, general-purpose computer. It belongs to the class of computers containing, among others, the IBM 360/20, IBM System 3, NCR Century 100 and 200, and the Univac 9300. However, the B1700 differs from the others in that (a) it is dynamically micro-programmable, and (b) it is designed to support hundreds of independent, special-purpose machine architectures, rather than one general-purpose architecture.

Each particular machine architecture is realized on a vertically micro-programmable B1700 processor by means of multiprogrammed interpreters. The general philosophy of the system is that each language that runs on the machine will have its own interpreter; i.e., the B1700 can be a "COBOL machine", a "FORTRAN machine", a "SNOBOL machine", an "APL machine", etc.

In keeping with this flexibility, a language (along with its interpreter) was designed to be used for implementation of the Master Control Program (MCP) and for implementation of the different compilers. This language is called the Software Development Language (or SDL).

SDL has so far been used to implement the MCP; compilers for SDL, COBOL, FORTRAN, BASIC, and the B1700 micro-language; and a sort package. Planned in the future are a Network Definition Language processor, an ALGOL compiler, and a Data Base Management System.

The purpose of this paper is to describe SDL and its underlying machine.

II. The SDL Language

A. History

The advantages of using a higher level language for system implementation are well documented in the literature (see Sammet, 1971; Corbato, 1969; or MIT, 1970). In fact, this use of a higher level language is merely in keeping with a Burroughs precedent (see Lyle, 1971).

Using input from the different software groups that would be using SDL, the SDL language and underlying machine structure were designed in the fall of 1969. In February, 1970, programming of a bootstrap version of the SDL compiler was begun by a four-man group working in Burroughs Extended ALGOL (Burroughs, 1969a) on the B5500. By June, 1970, a working version of the compiler, as well as a functional simulator of the SDL machine, were available on the B5500 for initial program checkout. Since then, The SDL language and machine have undergone several modifications, and the SDL compiler has been re-written in SDL by a two-man group to run on the B1700.

B. Language Form

The design philosophy of SDL was that it was to be "clean" and consistent (see Weinberg, 1971). Consequently, we attempted to avoid language features that:

1) require run-time routines to accomplish
2) are "niceties" that can actually be built from simpler features in the language (e.g., the DO-UNTIL statement)
3) we didn't feel we could implement well on a small machine
4) didn't "fit" (i.e., weren't needed to implement software)

The XPL language (McKeeman, 1970) appears to have excised from PL/I (Lucas, 1969) many of the PL/I features which fall into one of the above categories, and yet, retains those features which are best for compiler writing (see Slimich, 1971). Hence, SDL was designed using XPL as a base.

SDL is an ALGOL-like language. Allowable data types in SDL are bit strings, character strings, and fixed (integer) numbers, as well as single-dimensional arrays of these and structures of mixed data types. There are a number of excellent reasons for implementing a GO TO-free language (these are best summed in Weinberg, 1971; see also Dijkstra, 1968); and so SDL contains no GO TO's (neither does the SDL machine). Control is handled with IF-THEN and IF-THEN-ELSE statements, CASE statements, procedure invocations and returns, DO and DO FOREVER statements, and block-exit statements. Procedures in SDL are automatically recursive with up-level addressing. Run-time routines are needed only to handle virtual memory (when used). An SDL program consists of data

declarations, procedure declarations, and executable statements—in that order. An SDL procedure is a microcosm of an SDL program: it consists of a procedure head followed by data declarations, procedure declarations, and executable statements. A BNF description of the syntax of the language is included in the SDL Programmer's Reference Manual (Burroughs, 1972).

C. Data Structures

SDL data types are minimal but, nevertheless, are designed to provide for a wide range of needs with as little overhead as possible. Included are only those data types which are necessary for operating system and compiler development, and which we could implement well in a small-machine environment without run-time routine overhead penalties. Specifically excluded because of their inutility to software programming are floating point and decimal data types.

There are three types of data in SDL: bit strings (BIT), character strings (CHARACTER), and fixed (integer) numbers (FIXED). For example,

```
DECLARE
    A FIXED,
    B BIT(7),
    (C,D) CHARACTER(1023);
```

declares A to be an integer number, B to be a bit string of length 7 bits, and C and D to be character strings of length 1023 bytes each.

These basic data types may be grouped in structures, or single-dimensional arrays, or combinations of these. For example,

```
DECLARE
    01  A,
        02 A1(9) BIT(3),
        02 A2(3) FIXED,
        02 A3(7) CHARACTER(1);
```

declares A to be a structure whose sub-items are arrays. The example:

```
DECLARE
    01 B(7),
        02 B1 FIXED,
        02 B2 BIT(37),
            03 B21 BIT(34),
            03 B22 BIT(3),
        02 B3 Character(5);
```

declarés B to be an array, the elements of which are structures. The example:

```
DECLARE
    01 C BIT(81),
        02 C1 BIT(17),
        02 C2(5) CHARACTER(1),
        02 C3 FIXED;
```

declares C to be a structure, one sub-item of which is an array.

A data structure may be declared as a template in order that it may be applied to more than one data area. This is done with indexing combined with the "REMAPS BASE" declaration:

```
DECLARE
    01 AREA REMAPS BASE CHARACTER(40),
        02 AREA1 BIT(8),
        02 FILLER CHARACTER(30),
        02 AREA2 CHARACTER(9);
```

Data items may also re-describe other data items:

```
DECLARE
    A CHARACTER(80),
    B(80) REMAPS A CHARACTER(1);
```

describes an 80-byte data area as a single unit (A) and as an aggregate of single-byte pieces (B).

Simple dynamic data items, whose size is computed at run time:

```
DECLARE DYNAMIC C BIT(A*B-3);
```

can be used to avoid wasting unused bits. Although dynamic data items may not be structured, they may be re-described ("remapped") and thus provided with structure in this way.

Paged arrays allow the programmer to explicitly parameterize virtual storage:

```
DECLARE PAGED(64) D(1024) CHARACTER(500);
```

Here, D is a paged array of 1024 elements, each 500 bytes in length, with 64 elements per page. The SDL machine automatically keeps only as many pages in memory as will conveniently fit. Paging is on a demand basis.

In retrospect, it would be nice to have virtual strings; i.e., an invisible implementation of virtual memory. The execution-time penalty of providing this, however, was not outweighed by its advantages.

E. Statements

There are basically three types of statements in SDL: the assignment statement (considered to be an expression), the control statement (including conditional, group, and case statements, and procedure calls), and the function statement (including input-output statements and others).

Expressions

SDL expressions are fairly rich in nature, allowing IF-THEN-ELSE, CASE, and intermediate assignment, as well as arithmetic, logical, relational, and string operators. All data type combinations are permissible: There is no type conversion. In most cases, the data type is ignored; in assignments and comparisons, the data type is significant. For example, CHARACTER to CHARACTER comparison results in the shorter of the two operands being filled (functionally) on the right with blanks, whereas BIT to BIT comparison will cause zero fill to the left.

Group Statements

There are two means of grouping statements into a block: DO groups and DO FOREVER groups. Both DO groups and DO FOREVER groups may be named. DO groups may be exited by "falling out the bottom". DO FOREVER groups must be (and DO groups may be) explicitly exited through use of the UNDO statement.

As the name implies, DO FOREVER groups cycle back to the beginning of the group "forever", unless an UNDO or RETURN is executed. Several layers of nested groups may be exited by specifying in the UNDO statement

the name of the outermost group to be exited. An example follows:

```
DECLARE
    (IN(5),OUT(5),CARD(80)) CHARACTER(1)
   ,(I,C) BIT(24)
   ;
C←0;
DO SCAN.CARD FOREVER;
  I←0;
  DO COMPARE.TO.IN FOREVER;
       IF CARD(C)=IN(I) THEN
       DO;
            CARD(C)←OUT(I);
            UNDO COMPARE.TO.IN;
       END;
       IF 5=BUMP I THEN UNDO;
    END COMPARE.TO.IN;
    IF 80=BUMP C THEN UNDO;
END SCAN.CARD;
```

CASE Statement

The CASE statement has the form:

```
CASE ⟨expression⟩;
     ⟨statement 0⟩;
     ⟨statement 1⟩;
          .
          .
          .
     ⟨statement n⟩;
END CASE;
```

The ⟨expression⟩ must generate a value between 0 and n. This value is used to select one of the n+1 statements for execution. If the value is less than 0 or greater than n then a run-time error will occur.

Conditional Statements

The conditional statement can take either of the forms:

IF ⟨condition⟩ THEN ⟨statement⟩;

or:

IF ⟨condition⟩ THEN ⟨statement⟩;
ELSE ⟨statement⟩;

The ⟨condition⟩ may be any expression—however, only the low order bit is used: 0 as "false", 1 as "true".

Other control statements (e.g., FOR or DO...UNTIL) can often be fabricated using the definitional facility, described in Section II-G, below.

Function Statements

Input-output statements are included for the use of the compilers. There is neither a format nor a list, as such. The input-output statement has the syntax:

⟨I/O mode⟩ ⟨file name⟩ ⟨record key⟩ (⟨work area⟩);

where

⟨I/O mode⟩ ::= READ|WRITE
⟨file name⟩ ::= ⟨file identifier⟩
⟨record key⟩::= [⟨expression⟩] | ⟨empty⟩
⟨work area⟩ ::= ⟨"address-generating" expression⟩

No execution-time routines are invoked to effect input/output.

For a description of some of the other function statements, see II-J: Special Constructs.

F. Program Segmentation

Segmentation of SDL programs is entirely under the control of the programmer. It was felt that systems programmers would take the time and effort to segment their programs in as efficient manner as possible. In addition, the ability should exist to place into the same segment (or segment page) code which, although separated in space, is not separated in time.

Segmentation of SDL programs takes place at two levels: (1) placing code groups into segments, and (2) placing segments (actually segment pointers) into pages. The former is done principally with the SEGMENT statement:

```
SEGMENT(ERROR.ROUTINE);
```

which establishes (in this case) ERROR.ROUTINE as the name of the current segment. The latter is done with the SEGMENT.PAGE statement:

```
SEGMENT.PAGE(TAPE.ERROR OF IO.ERROR);
```

which establishes TAPE.ERROR as the current segment and IO.ERROR as the current page. The SEGMENT statement may change the current page.

There are two types of code segmentation effected by the SEGMENT statement: temporary and permanent. Temporary segment change occurs when the SEGMENT statement precedes a "subordinate" statement (i.e., the statement following THEN or ELSE, or a statement in a CASE statement). All other segmentation change is permanent. For example:

```
SEGMENT(X);
A←B;
IF C THEN
```

```
SEGMENT(Y);
DO;
    A←D;
    B←C;
END;
/* AT THIS POINT THE CURRENT SEGMENT AGAIN BECOMES "X"  */
```

As an SDL program executes, the SDL machine can collect usage statistics for each segment, thereby providing a dynamic feedback to the programmer on how well (or poorly) he has segmented his program (see II-I: Measurement and Debug Facilities).

G. Definitional (Macro) Facility

The advantages and importance of macro facilities have been described in Cheatham, 1966. SDL provides for both textual replacement (described here) and textual inclusion or exclusion (described in II-H). (Cheatham classifies both of these as "text macros"). The mechanism described here has previously appeared in Burroughs Extended ALGOL for the B5500 (see Burroughs, 1969a) and in Burroughs Extended ALGOL for the B6700 (see Burroughs, 1971). The SDL Definitional Facility has been quite heavily exploited in both the Master Control Program (MCP) and in the compilers.

The Definitional Facility allows symbols (actually tokens) in an SDL program to be replaced with other tokens or strings of tokens. For example:

```
DEFINE X AS #A+B#;
```

would cause every occurrence of X to be compiled as A+B.

Definitions can also be parametric; for example:

```
DEFINE X(N) AS #IF N THEN UNDO#;
```

The invocation X(A-B>C) would be compiled as:

```
IF A-B>C THEN UNDO;
```

Both define strings (the tokens between #'s) and define actual parameters may consist of many tokens, including other define invocations. For example:

```
DEFINE
    ESCAPE AS #SUCCESS←TRUE;  RETURN#,
    COMPARE(CS,S) AS
    #IF SYMBOL=CS THEN DO;  S; END#;
```

Then

```
    COMPARE("PAGE",WRITE PRINTER PAGE; ESCAPE);
```

would compile as

```
    IF SYMBOL="PAGE" THEN
    DO;
        WRITE PRINTER PAGE;
        SUCCESS←TRUE;
        RETURN;
END;
```

H. Conditional Compilation

The Conditional Compilation Facility of SDL provides a means for systematically including or excluding pieces of source code (in a program) depending on the setting of conditions. This facility is used most frequently to provide system extension. One may maintain a single source file for the MCP and include or exclude options (e.g., the Sort module, or the Data Communications module) at compile time. It is also heavily used to include or exclude debugging code. Optimized production systems and slow, self-checking systems can be generated and developed as a single program. The debugging code need never be physically removed from the source file, only conditionally excluded.

The conditional compilation facility provides a means of including (or excluding) source images depending upon the value of Boolean variables which may be set or reset at compile time.

The conditional compilation records contain an "&" in column 1, followed by a key word, followed by other symbols; the allowable statements are:

SET ⟨identifier list⟩

RESET ⟨identifier list⟩

IF ⟨Boolean expression⟩

ELSE

END

The ⟨Boolean expression⟩ is made up of identifiers which have appeared on a SET or RESET record and of the connectives AND, OR, and NOT. No

parenthesization is allowed.

Images which may be conditionally included or excluded are those which are delimited by IF-END, IF-ELSE, or ELSE-END. If the ⟨Boolean expression⟩ following an IF is true, then the images between the IF and its matching END or matching ELSE will be included in the compilation. Otherwise, the images between the matching ELSE and its END will be included. As implied, conditional inclusion groups may be nested. As an example:

```
    & SET A,B,C
    & RESET D
    & IF A
    X←0;
    & IF B AND NOT D
    X←1;
    & ELSE
    X←2;
    & END
    X←3;
    & END
    & IF B AND D
    X←4;
    & ELSE
    X←5;
    & END
```

would compile as:

```
    X←0;
    X←1;
```

```
X←3;
X←5;
```

I. Measurement and Debug Facilities

A number of measurement and debug facilities have been included in SDL to assist in MCP and compiler checkout, and to assist in system and program evaluation. In addition to those features described below, the definitional and conditional inclusion facilities have been very heavily exploited in providing "removable" debug and analysis code.

Dump

At any point within his program, the SDL programmer may specify that his program's data areas are to be dumped to a disk file for later analysis. There is a dump analysis program which can then be run, and which prints the descriptors and the data described by each descriptor.

Trace, Notrace

Since system checkout involved the MCP, interpreters, and compilers, as well as SDL programs, themselves, it proved expedient to include a facility whereby the program running, the MCP, or both could be traced. The TRACE command allows this, and also allows the specification of the type of trace for each: trace those commands which modify data items, trace those commands which change the Program Pointer Stack, trace all other commands, or any combination of the three. Needless to say, tracing is an interpreter function: since each program has its own interpreter (i.e., provides its own interpreter environment), tracing of a program does not affect any other program in the mix, including the MCP. NOTRACE turns off the tracing phenomenon.

The trace output may also be directed to magnetic tape or disk, for later programmatic analysis. One use that has been made of this capability is to locate the most frequently referenced pieces of code. Another is to analyze inter-segment branches: if two segments only reference each other, then the two segments may be merged, if the size of the conglomeration of the two is not too large. This branchpoint analysis has also indicated segments which are traversed frequently but contain little code, and therefore indicate that recoding (or re-segmentation) is needed.

Monitor

The HARDWARE.MONITOR instruction makes available on the backplane of the B1700 an 8-bit code which may then be sensed by a monitoring device. We are currently using the Computer Performance Monitor II, marketed by Allied Computer Technology Inc. In this case, the 8-bit code is used to turn timers on and off, bump counters, control counting periods, cause counters and timers to be dumped to magnetic tape for later analysis, etc.

Profile

The SDL programmer may specify at compile time that he wants statistics collected about selected parts of his program: he may count the number of entrances to selected procedures or he may count the number of times selected branch points are taken. At the end of execution of his program, the profile statistics are sorted and printed, thus giving the programmer a means of determining the "hot spots" in his program (candidates for

re-coding), the "cool spots" (code which may be moved to a less frequently referenced segment), and the "cold spots" (unused code which may indicate flaws in the programmers logic).

The program profile has also been useful in evaluating the SDL machine design: i.e., the selection of machine primitives. When a compiler function, such as scanning, shows up as a hot spot in all compilers, it is a clear indication that a new primitive should be added to the SDL machine. See II-J: Special Constructs (in particular, see Compiler Constructs).

J. Special Constructs

In writing an operating system or a compiler, one finds that there are special requirements that are unique to those applications of a language. In SDL, it has been necessary to provide these unique functions in a number of areas. Since these are relatively infrequently used functions, the interpreter code to effect the operators which provide these functions is normally not resident in control memory, but rather it exists in (and is executed from) main memory or has been overlayed to disk. Hence, a very low price is paid for making these extensions to the SDL language and machine. The advantages of having them far out-weigh the disadvantages.

MCP Constructs

There are functions which are unique to an operating system. In order to avoid the use of "in-line assembly language", special operators or function calls were included in SDL. These include:

1) Dispatch: causes the initiation of input/output operations
2) Memory Size: returns the sizes of M-memory and S-memory
3) Interrogate Interrupt Status: returns any interrupt bits which have been set since the MCP was entered
4) Search Linked List: used in the space allocation routine
5) Parity Address: used to search memory for (as yet) undetected parity errors
6) Fetch: fetches the results of an input/output operation
7) Reinstate: reinstates a user program
8) Overlay: causes overlaying of an interpreter
9) Enable/Disable Interrupts: allows or disallows interrupts
10) Return and Enable Interrupts: special return from the high

priority interrupt routine

Sort Constructs

A system sort procedure is typically one of those programs on which system performance is based. Consequently, it was felt that the most frequently performed sort functions should be done in special operators. The special constructs added to SDL for sort are:

1) Sort Step Down: provides the result of comparing two records using a table to provide the location and type of the comparison key
2) Sort Unblock: essentially does record unblocking, but will create tags rather than records if told to do so
3) Sort Search: provides the information to evaluate a record for sorting. The parameters provide the address of the first record to be examined and the condition(s) under which records will be selected
4) Initialize Vector: initializes the sort vector
5) Thread Vector: threads a new entry through the initial vector

Compiler Constructs

By analytical means it was discovered that all the compilers were spending some fairly large amount of time doing some similar functions. Hence, operators were designed which would be applicable to all (most) of the compilers on the B1700:

1) Hashcode: returns a hashcode based on the characters of the passed parameter
2) Deblank: removes blanks preceding a token

3) Next Token: returns the descriptor of the next token to be scanned

4) Delimited Token: returns the descriptor of the string of characters delimited by the specified character

Network Definition Language Constructs

The Network Definition Language processor (data communications) has unique requirements not usually found in other programs; part of these requirements are reflected by the operators:

1) Disperse/Retrieve: message access operators

2) Enter/Exit Coroutine: coroutine entrance and exit operators

K. Evaluation

A discussion of any language of this type would not be complete without some indication of the effectiveness of the language itself, and, in this case, some measure of the effectiveness of the implementation in a soft environment.

The reaction of the people using SDL has been a definite preference for SDL over other languages they have used, including ALGOL, ESPOL (see Burroughs, 1968), PL/I, and COBOL. In addition, there have been relatively few additions to the basic structure of the language: the notable exceptions have been dynamic data declarations, paged arrays, and a means of selecting from a structure only those descriptors needed on a given lexic level (to avoid Name Stack build-up).

The effectiveness of the implementation is probably best indicated by the amount of code generated per source statement. Since this statistic was not readily available, the amount of code generated per source image (card) will be used instead. This ranges from a low of 4.53 bytes per card for the SDL compiler itself to 5.11 for the MCP to 7.95 for the RPG compiler (which uses the definitional facility very heavily). The average for the MCP and all the compilers, as an aggregate, is 5.98 bytes of instruction per source image. If in-memory data space is included in this calculation, then the average is 6.51 bytes of space per source image. This compares very favorably with assembly language code requirements on the more popular byte-oriented machines, yet SDL is a higher-level language.

III. Overview of the B1700

The B1700 is a small, general-purpose computer (Burroughs, 1972b) that is particularly wellsuited for interpretation and emulation. The features of the B1700 that make it unique and unprecedented are:

1) Dynamically alterable, vertical microprogramming
2) Bit addressable main memory
3) Dynamic control of functional width of processor registers and busses
4) Dynamic control of memory access width
5) Microprogram subroutine capability
6) Stack structure

(For a more detailed discussion of the B1700, see Wilner, 1972a).

Principles first espoused in the Burroughs B5500 (Burroughs, 1969b) and in the Burroughs B3500 (Burroughs, 1969d) have culminated in the B1700. The B5500 is designed to process ALGOL, while the B3500 is a COBOL machine. Both of these machines have their designs hard-wired into them. The B1700, however, is "soft" at the level that the others are "hard". This, combined with a micro-order designed for interpreter writing, combined with the attributes listed above, have produced a machine that is singular in its capacities.

The virtual machines which have been produced for the B1700, including the SDL machine, are an order of magnitude more powerful at what they do than are hard-wired systems. Programs represented in these soft machine languages are from 25% to 75% smaller than on byte-oriented systems.

IV. The SDL Machine

The SDL language was designed to be used for implementation of the MCP and for implementation of the different compilers. In conjunction with the design of the language, was the design of a "machine" that would "execute" the statements of the language.

The SDL machine is a conglomeration of the ideas of many people. Particularly included are the language-directed design ideas of McKeeman (McKeeman, 1967), the stack and display mechanism of Randell and Russell (Randell, 1964), and the design of the Burroughs B6700 (see Hauck, 1968). See also Burroughs, 1969b and Burroughs, 1969c. The original SDL machine was designed by G. Brevier and B. Rappaport of Burroughs Corporation. Later additions and modifications to the basic machine design included ideas of C. Kaekel and the author, as well as other employees of Burroughs Corporation.

This section will describe the resulting S-machine and S-language.

A. Stack Mechanism

A B1700 program consists of code segments scattered in memory, one block of data bounded by a Base Register and a Limit Register, and a contiguous, read-only block (the Run Structure) containing program attributes. Also scattered throughout memory, in addition to code segments, are file attribute blocks and segment dictionaries. The area inside Base-Limit is divided into two parts: a static part and a dynamic part. In the case of an SDL program, the static area contains the S-machine stacks and the dynamic area contains paged array page tables and paged array pages (see Figure 1).

The SDL machine stack structure originally evolved from Randell and Russell (see Randell, 1964) and from the B6700 (see Hauck, 1968). This scheme has proved to be clean and easy to implement, and has resulted in a relatively small amount of code in the interpreter for stack management.

The structure of the S-machine stacks is shown in Figure 2a. The inter-relationships among the stacks are shown in Figure 2b. The Name Stack and the Program Pointer Stack run toward the Base Register (toward low memory addresses); the others run toward the Limit Register (high memory addresses). The stacks are used as follows:

1) Program Pointer Stack: This is a 32-bit wide stack that holds code addresses. Entries are pushed onto this stack upon procedure or DO group entrance, and are popped off upon procedure or DO group exit.

2) Control Stack: This is a 48-bit wide stack which maintains

the dynamic history of the allocation of data items. Entries are pushed onto this stack upon entrance to procedures with parameters and/or local data, and are popped off upon exit from these procedures.

3) Name Stack: This is a 48-bit wide stack that holds data descriptors. The data descriptors may contain values (self-relative) or the address of the values (in the Value Stack). Each lexic level's data descriptors occupy a contiguous block of entries in the Name Stack.

4) Value Stack: The Value Stack is a variable width stack which contains values of currently allocated (non self-relative) data items, as well as the values of temporary data items (i.e., intermediate values of expressions).

5) Evaluation Stack: The Evaluation Stack is a 48-bit wide stack which contains data descriptors for intermediate results and for temporary storage of procedure actual parameters.

6) Display: Display is a 32-bit wide array, the entries of which contain the addresses of the blocks of data allocated by the currently active lexic levels. The addresses in Display point into the Name Stack. A lexic level number is used to subscript into Display. In other words, Display points to all the groups of descriptors that can be currently addressed.

B. Opcode Structure

Because of SDL's stack structure and segmentation, code and data addresses are short, making the number of bits devoted to opcodes quite significant. In fact, more bits are used for opcode representation than for any other purpose, amounting to over one-third of a program's code space. Consequently, it was essential that not only should opcodes be represented in as compact manner as possible, but also that decode time for opcodes should be minimal.

The SDL S-operators use an encoding based on static frequence of occurrence. Operators are 4,6, or 10 bits in length with the most frequently occurring operators requiring the smaller number of bits.

The first 10 of the 4-bit codes (0_{16} through 9_{16}) represent operators. The next 5 are escape codes which indicate that the next 2 bits are to be examined in order to determine which operator is to be used. The last 4-bit code (F_{16}) is an escape code which indicates that the next 6 bits are to be used in order to determine which operator is to be used (see Figure 3).

Originally, the SDL S-operators were encoded using a 3-bit, 9-bit code. After a fairly large amount of working SDL code had been generated (in the MCP and the compilers) an analysis was done (on a static basis) of the operators used in that code in an attempt to verify that the proper encoding had been chosen, or, alternatively, to empirically arrive at one that would be optimal.

If Huffman's algorithm for minimum redundancy codes (see Huffman, 1952) had been used for SDL opcodes, the space requirements would have been minimal, but the time for decoding would have been large. A fixed field size would have minimized decoding time but would have required a large amount of storage. Using the opcode frequence obtained from the analysis mentioned above, an encoding was obtained that was very near the Huffman encoding in space required, but still small in decoding time (see Figure 4a,b).

Appendix I contains the SDL S-operators, along with their arguments and sizes. It is, perhaps, interesting to note that:

1) The operator associated with IF-THEN (IFTH) is a 4-bit operator while the operator associated with IF-THEN-ELSE (IFEL) is a 6-bit operator
2) All types of literals are used frequently enough to warrant 4-bit operators (ZOT, ONE, LITN, LIT)
3) Load Address (LA) is a 4-bit operator while Load Value (L) is a 6-bit operator. This result indicates (because of the way that the SDL expression code generator generates code) that there are more "simple" expressions than "complex" ones.
4) The operator (UNDO) for DO group and simple procedure exits is a 4-bit operator
5) Comparison for equal (EQL) and unequal (NEQ) are more frequently used than the other comparison operators (LSS, LEQ, GTR, GEQ)

For further description of B1700 memory utilization, see Wilner, 1972b.

C. Descriptor Formats

Each SDL data item is represented by a descriptor which specifies the attributes of that data item. The data attributes are thus contained in the data area, rather than being imbedded in the code. The implications of this are that there tend to be fewer instructions (for example, there is one add instruction for all possible types—including mixed types—rather than a bit add, a character add, a fixed add, etc.) and that the instructions tend to be more compact since they reference descriptors for attributes, rather than contain the attributes themselves.

Descriptors in SDL are of two types: simple variables and arrays (variables to be subscripted). Simple descriptors are 48 bits in length while array descriptors are 96 bits in length. (See Figure 5.)

Simple descriptors have a type field (discussed below), a length field, and a field which contains the data (if the data is not more than 24 bits in length and is not in a structure), or the address of the data (if the data is more than 24 bits in length or is in a structure).

Array descriptors have a type field, a field giving the length of each element, a field giving the address of the first element, a field giving the number of bits to truncate from the right of a subscript to obtain the page subscript (paged arrays only), a field giving the length between elements (this is equal to the length of the element on the lowest level only of a structured array), and a field giving the number of elements in the array.

The bits in the type field (see Figure 5) are used as follows:

Bit	Use
0	1 if the value has been loaded to the top of the Value Stack (used when the descriptor is on the Evaluation Stack only); 0 otherwise
1	1 if descriptor is non self-relative; 0 otherwise (data item is in address field)
2	1 if array descriptor; 0 if simple descriptor
3	1 if length of element equals length between elements; 0 otherwise (arrays only)
4,5	Data type; BIT (00), FIXED (01), CHARACTER (10), VARYING (11) (formal descriptors only)
6	1 if paged array; 0 otherwise (arrays only)
7	1 if length varying (formal descriptors only)

It should be pointed out that the use of descriptors along with the bit-addressability of the B1700 allows a greater variety of data representations, so that the extra bits are more than made up for by not having to use "unnatural" representations (a byte for a one-bit flag, for example).

D. Code Addressing

All code on the B1700 is not only re-entrant, but also automatically relocatable, since code addressing is done through code pointers (segment Dictionary entries), rather than with memory addresses (this is necessary for re-entrancy when the code is overlayable, but not sufficient: see IV-E, Data Addressing). The MCP and compilers tend to be large programs and, hence, have a large number of segments since the segments themselves must be small (due to the memory restrictions of the B1700). In addition, in procedure-oriented languages such as SDL, and in compilers in particular, programs are written in"passes" (this is also true for the MCP, to some extent: the collection of procedures to process control cards, for example, or the procedures to process I/O error conditions). In other words, code which is executed together in time is gathered into segments, and segments which are executed together in time are gathered together into pages. Thus, SDL code addresses specify (either explicitly or implicitly) a triple that is used to generate an actual memory address if the segment is present, or a disk address if the segment is missing from memory.

Code addresses in the SDL machine actually appear as pairs, triplets, or quadruplets (Figure 6).

The Type field indicates the presence or absence of the Segment Number field and of the Page Number field, as well as the size of the Displacement field. The Page Number is the entry in the master Segment Dictionary used to find the minor Segment Dictionary to be used (if the minor Segment

Dictionary is not present, then an interrupt is generated). The Segment Number is used to locate the entry in the minor Segment Dictionary which gives the location of the desired segment (if the segment is not present, then an interrupt is generated). The Displacement gives the relative offset into the segment of the instruction being referenced.

This encoding allows the SDL machine to directly address 2^{30} bits of code. This yields a 38.4% savings in space for the SDL machine when compared to a byte-oriented machine with equal addressing capability (see Wilner, 1972b).

E. Data Addressing

SDL data addresses are two-part addresses, the first part specifying the lexic level of declaration of the data item, and the second part specifying the occurrence number of the data item within that lexic level. The data addresses do not contain memory addresses: this is the second condition that is necessary for re-entrancy. It also allows SDL procedures to be automatically recursive, and is part of the up-level addressing scheme.

SDL data addresses are three-part addresses (see Figure 7). The Type field specifies the size (and type of contents) of the two following fields. The lexic level field indicates which entry of Display to use to subscript into the Name Stack. The occurrence number field is the number of 48-bit descriptors to offset to find the indicated descriptor. If Display and the Name Stack are considered as arrays, and V(LL,ON) is the address represented by a Type, Lexic Level, Occurrence Number triple, then

V(LL,ON)=NAME.STACK(DISPLAY(LL)+ON)

represents the formula used to calculate an address in the Name Stack.

F. Descriptor Construction Operators

As a procedure (lexic level) is entered, the local data for that lexic level is created by entering onto the Name Stack the descriptors for the local data. The descriptors are constructed with operators.

Rather than carry the descriptors intact in the code or somewhere else in memory, they are carried, in an encoded form, in-line behind the operators which describe how the address field of the descriptor is to be derived. The in-line descriptor format and the Construct Descriptor Operators and their arguments are shown in Figure 8a. The formulae for descriptor address calculations are shown in Figure 8b.

The action of each of the operators is as follows:

Construct Descriptor Base Zero (CDBZ): A descriptor is put on the Name Stack with an address of zero.

Construct Descriptor Local Data (CDLD): The number of descriptors specified are constructed using the current value of the Value Stack Pointer as the address. The Value Stack Pointer is kept current as each descriptor is put on the Name Stack by adding to the Value Stack Pointer the length of the data item described.

Construct Descriptor From Previous (CDPR), Construct Descriptor From Previous and Add (CDAD), Construct Descriptor From Previous and Multiply (CDMP): The number of descriptors specified are constructed using the following formulae to calculate the addresses:

CDPR: A'=A+F

CDAD: A'=A+F+L

CDMP: A'=A+F+L+(E-1)xLB

where

A' is the new address part

A is the address part of the previous entry in the Name Stack

F is the in-line filler value if present

L is the length part of the previous entry on the Name Stack

E is the number-of-entries part of the previous entry on the Name Stack

LB is the length-between part of the previous entry on the Name Stack

Note that CDMP assumes that the previous entry on the Name Stack is an array descriptor.

Construct Descriptor Lexic Level (CDLL): A descriptor is constructed on the Name Stack which has as its address part the address of the value described by the descriptor specified by the LL, ON part.

These 6 operators are sufficient to construct all the descriptors required by all possible combinations of arrays, structures, and filler as described in II-C.

G. Handling of Control Statements

SDL's sophisticated segmentation allows segment changes to appear virtually anywhere within SDL programs. This non-sequential program flow combined with the lack of a GO TO in the S-machine created some interesting complexities. In an attempt to handle all of these complexities in a uniform manner, very heavy use was made of the Program Pointer Stack. All of the control statement operators (except Cycle) cause insertion or removal of entries from this stack. All of these operators can or do affect the next instruction address. The format of the control statement operators is given in Figure 9. A description of the operators follows.

Call (CALL): The Call operator is used to enter DO and DO FOREVER groups when these do not follow THEN and ELSE, and are not part of a CASE. The argument of the Call is the code address of the DO or DO FOREVER. Execution of the Call causes the current program address to be pushed onto the Program Pointer Stack, and the next instruction to be executed from the address indicated by the argument.

If-Then (IFTH): The If-Then operator is (as might be expected) used to handle the IF-THEN statement. The operator examines the low-order bit of the value described by the descriptor on the top of the Evaluation Stack. If this bit is 1 then the current program address is pushed onto the Program Pointer Stack, and the next instruction to be executed is taken from the address indicated by the (code address) argument.

If-Then-Else (IFEL): The If-Then-Else operator is used to handle the IF-THEN-ELSE statement. The current program address is pushed onto the Program Pointer Stack. If the low-order bit of the value described by the descriptor on the top of the Evaluation Stack is 1, then the next instruction address is indicated by the first code address following the operator; otherwise, the next instruction address is indicated by the second code address following the operator.

Case (CASE): The Case operator is used for CASE statements. The value described by the descriptor on the top of the Evaluation Stack is compared to the number, N, of code addresses following the operator: if the value is greater than or equal to N, then an error occurs; otherwise, the value is used to subscript into the code addresses. If the code address selected is null, then the operator is terminated and the next instruction is executed; otherwise, the current program address is pushed onto the Program Pointer Stack and the selected code address is used to obtain the next instruction address.

Undo (UNDO): UNDO statements are handled by the Undo operator. Since more than one level of nesting may be undone by any given UNDO statement, the number of levels to undo is contained in the instruction. The number of levels specified is popped from the Program Pointer Stack and the last one popped is used as the address of the next instruction.

Undo Conditionally (UNDC): The statement

IF ⟨condition⟩ THEN UNDO;

is one that causes needless manipulation of the Program Pointer

Stack if handled with the If-Then and Undo operators. Consequently, a special operator was devised which is no more than the amalgamation of these operators: if the low-order bit of the value described by the descriptor on the top of the Evaluation Stack is 1, then an Undo operation is performed; otherwise, the next instruction is executed.

Cycle (CYCL): DO FOREVER loops are handled by the Cycle operator. Since DO (and DO FOREVER) groups are required to terminate in the segment in which they began, it is sufficient to subtract some amount from the current program address. The amount to be subtracted is contained in the field following the Cycle operator.

It might be noted that, because some of these operators contain code addresses, it is possible to obtain some nice optimizations. In particular, if UTP is the name of an untyped procedure which has no parameters, then the following cases may be optimized by merely using the address of the procedure as part of the instruction;

```
IF ⟨condition⟩ THEN UTP;
IF ⟨condition⟩ THEN ...;  ELSE UTP;
CASE ⟨expression⟩;
    .
    .
    .
    UTP;
    .
    .
    .
END CASE;
```

H. Procedure Entrance and Exit

Procedure entrance and exit are a form of control statement execution, but are more complex than those statements described in IV-G, since the Control Stack and the Display may also be affected.

Procedure entrance and exit always affects the Program Pointer Stack and affect the Control Stack and Display when there is local data and/or parameters.

A call to a procedure with no local data and no parameters requires only the Call operator (see IV-G). A call to a procedure with local data but no parameters requires a Call operator followed by a Mark Stack and Update operator executed inside the procedure. A procedure with parameters and with or without local data requires a Mark Stack operator, followed by the operators to put the actual parameters on the Evaluation Stack, followed by a Call operator. Inside the procedure, a Construct Descriptor Formal operator is executed. (See Figure 10).

The Call, Mark Stack, and Mark Stack and Update operators will be described here; the Construct Descriptor Formal operator will be described in section IV-I.

Call (CALL): The argument of the Call is the code address of the procedure to be entered. Execution of the Call causes the current program address to be pushed onto the Program Pointer Stack, and the next instruction to be executed from the address indicated by the argument.

Mark Stack (MKS): The Mark Stack operator causes construction of an entry on the top of the Control Stack. This entry contains the current values of the Name and Value Stack Pointers. The Exited Lexic Level field of the entry is set to the value of the current lexic level, and the Entered Lexic Level field is set to zero.

Mark Stack and Update (MKU): The Mark Stack and Update operator has as an argument the lexic level of the procedure being entered. This operator causes construction of an entry on the top of the Control Stack. The entry contains the current values of the Name and Value Stack Pointers. The Exited Lexic Level field of the entry is set to the value of the current lexic level, and the Entered Lexic Level field is set to the value specified as the operator argument. The Display Stack entry for the specified lexic level is set to the current value of the Name Stack Pointer. The current lexic level is changed to the specified lexic level.

All procedure exits are done with the RETURN statement; however, the operator generated depends upon whether or not the procedure contains local data or parameters, and upon whether or not the procedure is typed.

If the procedure contains no local data and has no parameters (and therefore did not change the Control Stack upon entrance), then an Undo operator is used to effect the return. If there is either local data or parameters and the procedure is not typed, then an Exit operator is used. If there is either local data or parameters and the procedure is typed, then a Return operator is used. (See Figure 10.)

The Undo operator was described in IV-G. The Exit operator will be described here, and the Return operator will be described in section IV-I.

Exit (EXIT): The Name and Value Stack Pointers are set to the values obtained from the top entry of the Control Stack. The Display entry pointed to by the current lexic level is restored to the Name Stack value obtained from the first (proceeding from top to bottom) Control Stack entry, if any, having an Entered Lexic Level field equal to the current lexic level (unless a prior or the present entry has a zero Exited Lexic Level field). The Exited Lexic Level field is used to set the current lexic level, and the top entry is popped from the Control Stack. The number of levels specified is popped from the Program Pointer Stack and the last one popped is used as the address of the next instruction.

I. Parameter Passing—Returning of Values

The formal parameter statement assigns a type (and length) to each of the formal parameters. The SDL programmer has the option of having the SDL machine (interpreter) verify that the actual parameter matches the formal parameter. Since this check is time-comsuming, it is typically not performed once a program has been debugged. The consistency check is performed by the Construct Descriptor Formal operator (see Figure 11). When the check is to be done, this operator has, as its arguments, "descriptor templates" for each of the formal parameters. The description of this operator follows:

> Construct Descriptor Formal (CDFM): The Construct Descriptor Formal operator assumes that a Mark Stack operator was executed before the actual parameters were placed on the Evaluation Stack. The current lexic level is changed to the lexic level specified by the operator. The specified lexic level is also put into the Entered Lexic Level field of the top entry in the Control Stack. The Display Stack entry for the specified lexic level is set to the current value of the Name Stack Pointer. The current lexic level is set to the specified lexic level. The number of descriptors specified is constructed on the Name Stack using the in-line descriptor information plus the corresponding descriptor information on the Evaluation Stack. The type and length fields are compared for consistency between corresponding descriptors on the Evaluation and Name Stacks. The Evaluation Stack is cut back after construction of the descriptors; the Value Stack is not.

The values returned by typed procedures in SDL should agree in type and length with the formal type of the procedure itself. The SDL programmer again has the option of specifying whether or not this consistency check is performed by the interpreter. If this check is to be performed then the Return operator contains a descriptor template in-line following the operator.

Return (RTRN): The Return is the same as the Exit operator prior to popping entries off the Program Pointer Stack. At this point, the data descriptor on the Evaluation Stack is compared to the in-line descriptor for consistency. If the data is on the Value Stack, then after cutting back the Value Stack, the data is moved to the new top of the Value Stack. The number of levels specified is popped from the Program Pointer Stack and the last one popped is used as the address of the next instruction.

J. Special Operators

In order to illustrate further the complexity and flexibility possible with a machine such as the B1700, several of the special operators will also be described.

Search Linked List

The Search Linked List operator is used principally by the MCP to allocate memory space. This operator compares a value with a list of linked structures, searching for the indicated relationship or the end of the list. The argument specifies the compare type: less, less or equal, equal, not equal, greater or equal, greater. There are four descriptors on the Evaluation Stack. The descriptors represent:

1) Link Index: the relative offset in the structure, and the size, of the field which contains the address of the next structure to be examined
2) Compare Variable: the variable to be compared to the linked structure
3) Argument Index: the relative offset in the structure, and the size, of the field to which Compare Variable is to be compared
4) Record Address: the address of the first structure to examine

The operator returns the address of the structure whose compare field was in the desired relationship to the Compare Variable, or it returns an indicator that there were no structures in the desired relationship.

Reinstate

The Reinstate operator is the operator used by the MCP to reinstate a

user program. The descriptor on the top of the Evaluation Stack is assumed to describe a field in the Run Structure of the program to be reinstated. The reinstating program's M-machine state is stored in its own Run Structure (each program currently executing has a Run Structure which contains the program's execution attributes). The address of the reinstating program's Run Structure is stored in the reinstated program's Run Structure. The descriptor at the top of the Evaluation Stack is removed. The address field of this descriptor addresses the Run Structure of the program which is then reinstated.

Next Token

The Next Token operator is used by compilers to scan source images. The first argument is the data address of a descriptor which describes the first character to be examined. It is assumed that this character is non-blank. The second argument is a "separator" character (such as "-" in COBOL). The third argument is the "numeric-to-alpha indicator".

If the character described by the first argument is a special character, then the operator is exited with a descriptor on the top of the Evaluation Stack which describes this character, and with the descriptor described by the first argument advanced to point to the next character in the source image.

If numeric-to-alpha indicator is 1 then the stopper is set to "A"; otherwise, if the first character is numeric then the stopper is set to "0"; otherwise, the stopper is set to "A". Characters are sequentially compared to the stopper until one is found which is less than the stopper

and not equal to the separator. The operator then exits with a descriptor on the top of the Evaluation Stack which describes the token just found, and with the descriptor described by the first argument advanced to point to the next character in the source image. (The EBCDIC collating sequence is assumed).

V. Conclusion

In this brief description of the B1700 Software Development Language (SDL), and its underlying S-machine, I have attempted to give some indication of the flavor of SDL but, more importantly, to illustrate the extreme flexibility and suitability of the B1700 for the tasks for which it was designed: the writing of (language) interpreters and emulators. We who have used SDL feel that it is well-suited for the type of programming for which it was designed. We could not agree more with Saltzer et al (MIT, 1970) that one of our best decisions was to program the operating system in a higher-level language. However, the degree of success of the software depends very heavily upon the suitability of the hardware to the software and to the language in which the software is written. The Burroughs B1700, by its very nature, has proven to be quite suitable to the tasks to which it has been assigned. It should be pointed out, that because all of the software for the B1700 has been written in a higher-level language, all of it (including the MCP) is theoretically transportable to any other system which has soft interpretation (of the flexibility of the B1700).

VI. Acknowledgements

This paper would be incomplete without acknowledgement to the people who are responsible for the original design of the SDL language and the SDL machine: G. Brevier, C. Kaekel, and B. Rappaport, all of Burroughs Corporation. Thanks also goes to W. Wilner for review and critique of this paper, and for analysis and evaluation of the SDL design.

VII. APPENDIX I: SDL S-OPERATORS

RELATIONAL OPERATORS

NAME	MNEMONIC	OP CODE SIZE	ARGUMENTS
EQUAL TO	EQL	6	
LESS THAN	LSS	10	
LESS THAN OR EQUAL TO	LEQ	10	
GREATER THAN	GTR	10	
GREATER THAN OR EQUAL TO	GEQ	10	
NOT EQUAL TO	NEQ	6	

ARITHMETIC OPERATORS

NAME	MNEMONIC	OP CODE SIZE	ARGUMENTS
ADD	ADD	6	
SUBTRACT	SUB	6	
MULTIPLY	MUL	10	
DIVIDE	DIV	10	
MODULO	MOD	10	
REVERSE SUBTRACT	RSUB	10	
REVERSE DIVIDE	RDIV	10	
REVERSE MODULO	RMOD	10	
NEGATE	NEG	10	
CONVERT TO DECIMAL	DEC	10	
CONVERT TO BINARY	BIN	10	

LOGICAL OPERATORS

NAME	MNEMONIC	OP CODE SIZE	ARGUMENTS
AND	AND	10	
OR	OR	10	
EXCLUSIVE-OR	XOR	10	
NOT	NOT	10	

STRING OPERATORS

NAME	MNEMONIC	OP CODE SIZE	ARGUMENTS
CONCATENATE	CAT	6	
SUBSTRING TWO	SS2	10	
SUBSTRING THREE	SS3	6	

SDL S-OPERATORS (CONTINUED)

LOAD OPERATORS

NAME	MNEMONIC	OP CODE SIZE	ARGUMENTS
MAKE DESCRIPTOR	MDSC	10	
VALUE DESCRIPTOR	VDSC	10	
DESCRIPTOR	DESC	6	DA
NEXT OR PREVIOUS ITEM	NPIT	10	V,DA
LOAD VALUE	L	6	DA
LOAD ADDRESS	LA	4	DA
ARRAY LOAD VALUE	AL	10	DA
ARRAY LOAD ADDRESS	ALA	6	DA
INDEXED LOAD VALUE	IL	10	DA
INDEXED LOAD ADDRESS	ILA	4	DA
LOAD LITERAL	LIT	4	D,LITERAL
LOAD 10-BIT LITERAL	LITN	4	LITERAL
LOAD LITERAL ZERO	ZOT	4	
LOAD LITERAL ONE	ONE	4	

STACK OPERATORS

NAME	MNEMONIC	OP CODE SIZE	ARGUMENTS
BUMP VALUE STACK POINTER	BVSP	10	
DUPLICATE	DUP	6	
DELETE	DEL	10	
EXCHANGE	XCH	6	
FORCE VALUE STACK	FVS	6	

STORE OPERATORS

NAME	MNEMONIC	OP CODE SIZE	ARGUMENTS
STORE DESTRUCTIVE	STOD	4	
STORE NON-DESTRUCTIVE LEFT	SNDL	6	
STORE NON-DESTRUCTIVE RIGHT	SNDR	10	

CONSTRUCT DESCRIPTOR OPERATORS

NAME	MNEMONIC	OP CODE SIZE	ARGUMENTS
CONSTRUCT DES. BASE ZERO	CDBZ	10	D
CONSTRUCT DES. LOCAL DATA	CDLD	6	N,D1,...,DN
CONSTRUCT DES. FORMAL	CDFM	10	LL,E
CONSTRUCT DES. FORMAL-V2	CDFM:	10	LL,E,D1,...,DN

SDL S-OPERATORS (CONTINUED)

LOAD OPERATORS

NAME	MNEMONIC	OP CODE SIZE	ARGUMENTS
MAKE DESCRIPTOR	MDSC	10	
VALUE DESCRIPTOR	VDSC	10	
DESCRIPTOR	DESC	6	DA
NEXT OR PREVIOUS ITEM	NPIT	10	V,DA
LOAD VALUE	L	6	DA
LOAD ADDRESS	LA	4	DA
ARRAY LOAD VALUE	AL	10	DA
ARRAY LOAD ADDRESS	ALA	6	DA
INDEXED LOAD VALUE	IL	10	DA
INDEXED LOAD ADDRESS	ILA	4	DA
LOAD LITERAL	LIT	4	D,LITERAL
LOAD 10-BIT LITERAL	LITN	4	LITERAL
LOAD LITERAL ZERO	ZOT	4	
LOAD LITERAL ONE	ONE	4	

STACK OPERATORS

NAME	MNEMONIC	OP CODE SIZE	ARGUMENTS
BUMP VALUE STACK POINTER	BVSP	10	
DUPLICATE	DUP	6	
DELETE	DEL	10	
EXCHANGE	XCH	6	
FORCE VALUE STACK	FVS	6	

STORE OPERATORS

NAME	MNEMONIC	OP CODE SIZE	ARGUMENTS
STORE DESTRUCTIVE	STOD	4	
STORE NON-DESTRUCTIVE LEFT	SNDL	6	
STORE NON-DESTRUCTIVE RIGHT	SNDR	10	

CONSTRUCT DESCRIPTOR OPERATORS

NAME	MNEMONIC	OP CODE SIZE	ARGUMENTS
CONSTRUCT DES. BASE ZERO	CDBZ	10	D
CONSTRUCT DES. LOCAL DATA	CDLD	6	N,D1,...,DN
CONSTRUCT DES. FORMAL	CDFM	10	LL,E
CONSTRUCT DES. FORMAL-V2	CDFM	10	LL,E,D1,...,DN

SDL S-OPERATORS (CONTINUED)

CONSTRUCT DESCRIPTOR OPERATORS (CONTINUED)

NAME	MNEMONIC	OP CODE SIZE	ARGUMENTS
CONSTRUCT DES. FROM PREV.	CDPR	6	N,D1,...,DN
CONSTRUCT DES. FROM PREV. & ADD	CDAD	6	N,D1,...,DN
CONSTRUCT DES. FROM PREV. & MULTIPLY	CDMP	10	N,D1,...,DN
CONSTRUCT DES. LEXIC LEVEL	CDLL	10	DA,D

PROCEDURE OPERATORS

NAME	MNEMONIC	OP CODE SIZE	ARGUMENTS
CALL	CALL	4	CA
IF THEN	IFTH	4	CA
IF THEN ELSE	ITEL	6	TYPE,CA,CA
CASE	CASE	10	N,TYPE,CA1,...,CAN
UNDO	UNDO	4	L
UNDO CONDITIONALLY	UNDC	10	L
RETURN-V1	RTRN	10	L
RETURN-V2	RTRN	10	L,D
EXIT	EXIT	6	L
CYCLE	CYCL	6	DISPLACEMENT
MARK STACK	MKS	6	
MARK AND UPDATE	MKU	10	LL

MISCELLANEOUS OPERATORS

NAME	MNEMONIC	OP CODE SIZE	ARGUMENTS
SWAP	SWAP	10	
INTERRUPT STATUS	IIS	10	
FETCH	FECH	10	
DISPATCH	DISP	10	
HALT	HALT	10	
READ CASSETTE	RDCS	10	
LENGTH	LENG	10	
LOAD SPECIAL	LSP	10	V
CLEAR	CLR	10	
COMMUNICATE	COMM	10	
REINSTATE	REIN	10	
FETCH CMP	FCMP	10	
ADDRESS	ADDR	10	
SAVE STATE	SVST	10	
HARDWARE MONITOR	HMON	10	
OVERLAY	OVLY	10	
PROFILE	PRFL	10	N
SEARCH LINKED LIST	SLL	10	V

REFERENCES

Burroughs, 1968 — Burroughs B5500 ESPOL Reference Manual, 1032638

Burroughs, 1969a — Burroughs B5500 Extended ALGOL Reference Manual, 1028024

Burroughs, 1969b — Burroughs B5500 Systems Reference Manual, 1021326

Burroughs, 1969c — Burroughs B6700 System Reference Manual, 1043676

Burroughs, 1969d — Burroughs B2500 and B3500 Systems Reference Manual, 1025475

Burroughs, 1971 — Burroughs B6700 Extended ALGOL Language Information Manual, 5000128

Burroughs, 1972a — Burroughs Small Systems Software Development Language Manual, (to be released)

Burroughs, 1972b — Burroughs B1700 Systems Reference Manual, 1057155

Cheatham, 1966 — Cheatham, T. E., Jr., "The Introduction of Definitional Facilities into Higher Level Programming Languages", Proc. FJCC, Vol. 29 (1966)

Corbato, 1969 — Corbato, F. J., "PL/I as a Tool for System Programming", Datamation, Vol. 15, No. 5, (May, 1969)

Dijkstra, 1968 — Dijkstra, Edsger W., "Go To Statement Considered Harmful", CACM, Vol. 11, No. 3, (March, 1968: Letters to the Editor)

Hauck, 1968 — Hauck, E. A., and Dent, B. A., "Burroughs' B6500/B7500 Stack Mechanism", Proc. SJCC, Vol. 32 (1968)

Huffman, 1952 — Huffman, D. A., "A Method for the Construction of Minimum Redundancy Codes", Proc. IRE, Vol. 40 (1952)

Lucas, 1969 — Lucas, P., and Walk, K., "On the Formal Description of PL/I", Annual Review in Automatic Programming, Vol. 6, Part 3 (1969)

Lyle, 1971 — Lyle, Don M., "A Hierarchy of High Order Languages for Systems Programming", Proc. ACM SIGPLAN Symposium on Languages for Systems Implementation, SIGPLAN Notices, Vol. 6, No. 9 (October, 1971)

McKeeman, 1967 — McKeeman, W. M., "Language Directed Computer Design", Proc. FJCC, Vol. 31 (1967)

McKeeman, 1970 — McKeeman, W. M., Horning, J. J., and Wortman, D. B., A Compiler Generator, Prentice Hall, Inc., Englewood Cliffs, N. J. (1970)

MIT, 1970 — Progress Report VII, Project MAC, Massachusetts Institute of Technology, Cambridge, Mass., p. 6 (July 1969 to July 1970)

Randall, 1964 — Randall, B., and Russell, L. J., ALGOL 60 Implementation, Academic Press, London (1964)

Sammett, 1971 — Sammet, Jean E., "A Brief Survey of Languages Used in Systems Implementation", Proc. ACM SIGPLAN Symposium on Languages for Systems Implementation, SIGPLAN Notices, Vol. 6, No. 9 (1971)

Slimick, 1971 — Slimick, John, "Current Systems Implementation Languages: One User's View", Proc. ACM SIGPLAN Symposium on Languages for Systems Implementation, SIGPLAN Notices, Vol. 6, No. 9 (1971)

Weinberg, 1971 — Weinberg, Gerald M., The Psychology of Computer Programming, Von Nostrand Reinhold Company, New York (1971)

Wilner, 1972a — Wilner, W. T., "Design of the B1700", Proc. FJCC, Vol. 41 (1972)

Wilner, 1972b — --------, "B1700 Memory Utilization", op. cit.

SDL MEMORY STRUCTURE

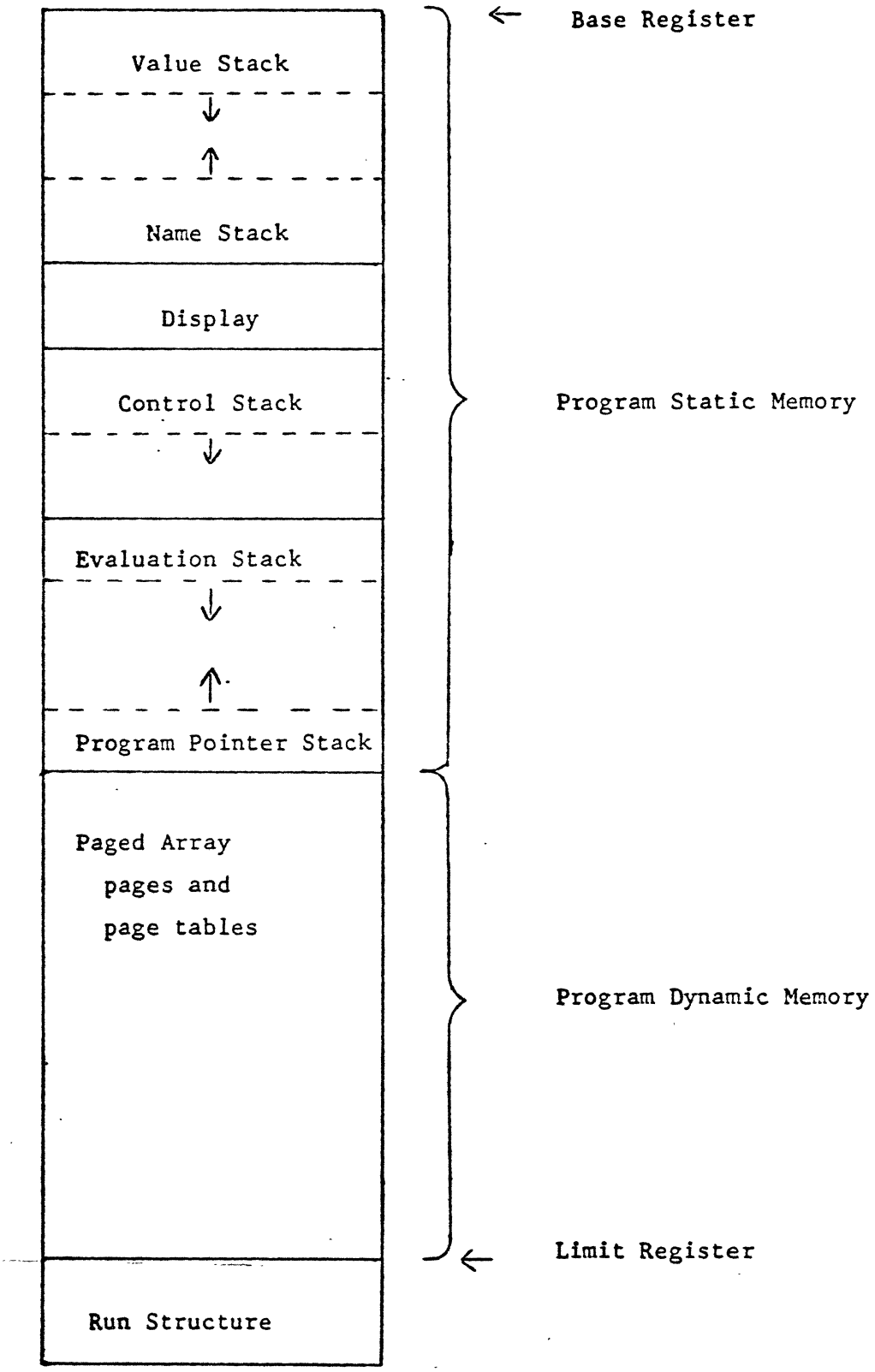
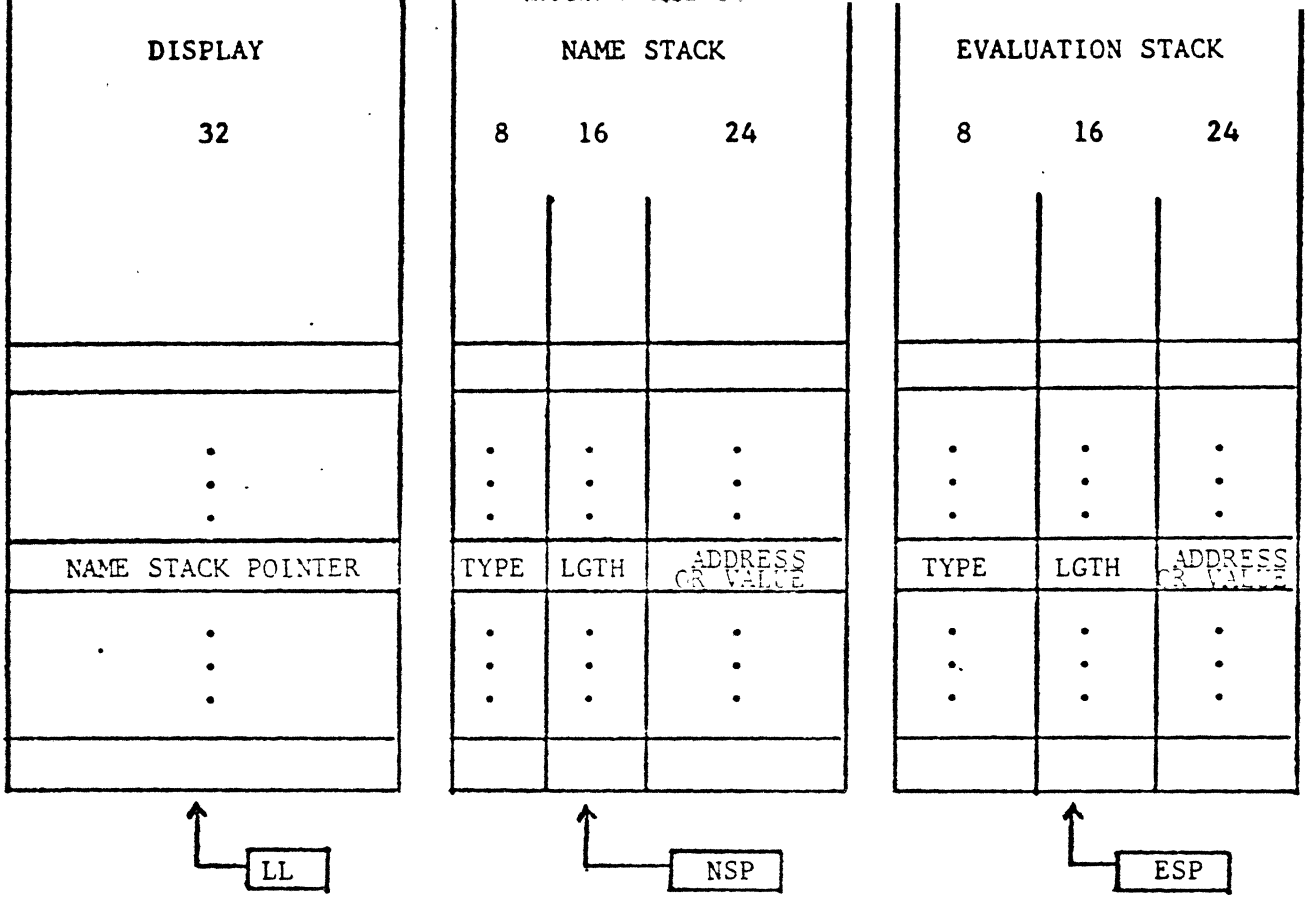
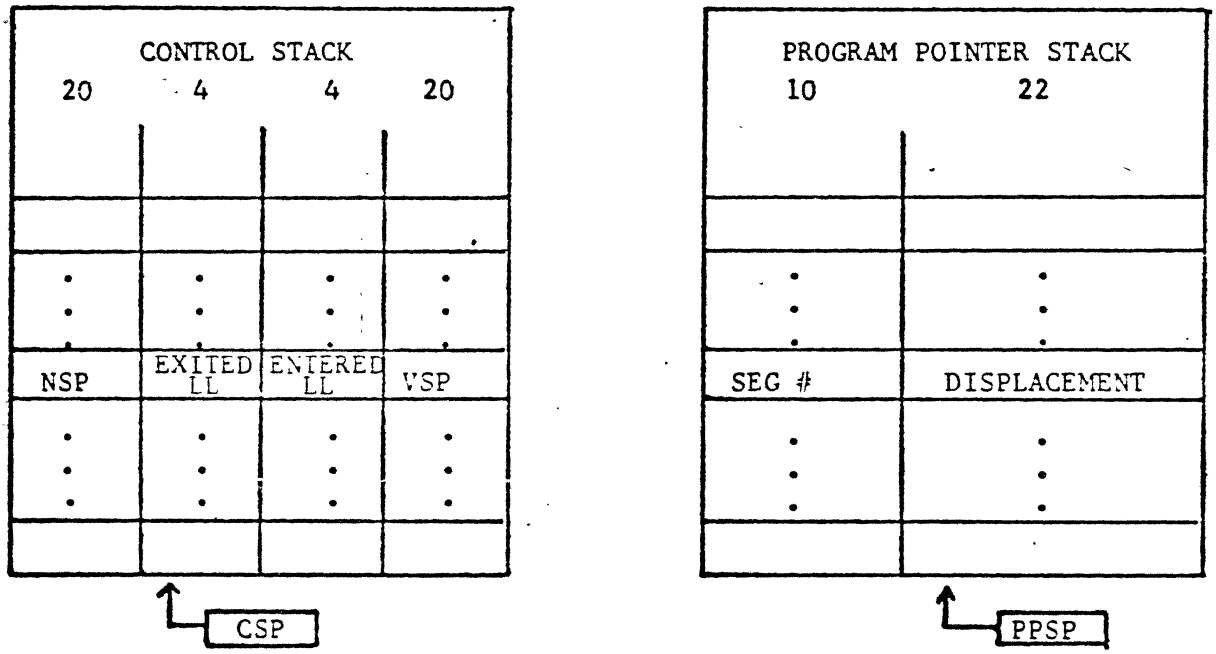


FIGURE 1.


DISPLAY
32
NAME STACK POINTER
LL
NAME STACK
8
16
24
TYPE
LGTH
ADDRESS OR VALUE
NSP
EVALUATION STACK
8
16
24
TYPE
LGTH
ADDRESS OR VALUE
ESP


CONTROL STACK
20
4
4
20
NSP
EXITED LL
ENTERED LL
VSP
CSP
PROGRAM POINTER STACK
10
22
SEG #
DISPLACEMENT
PPSP

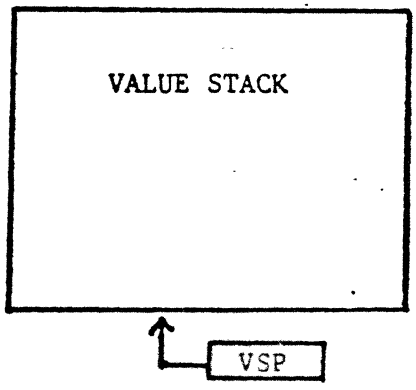
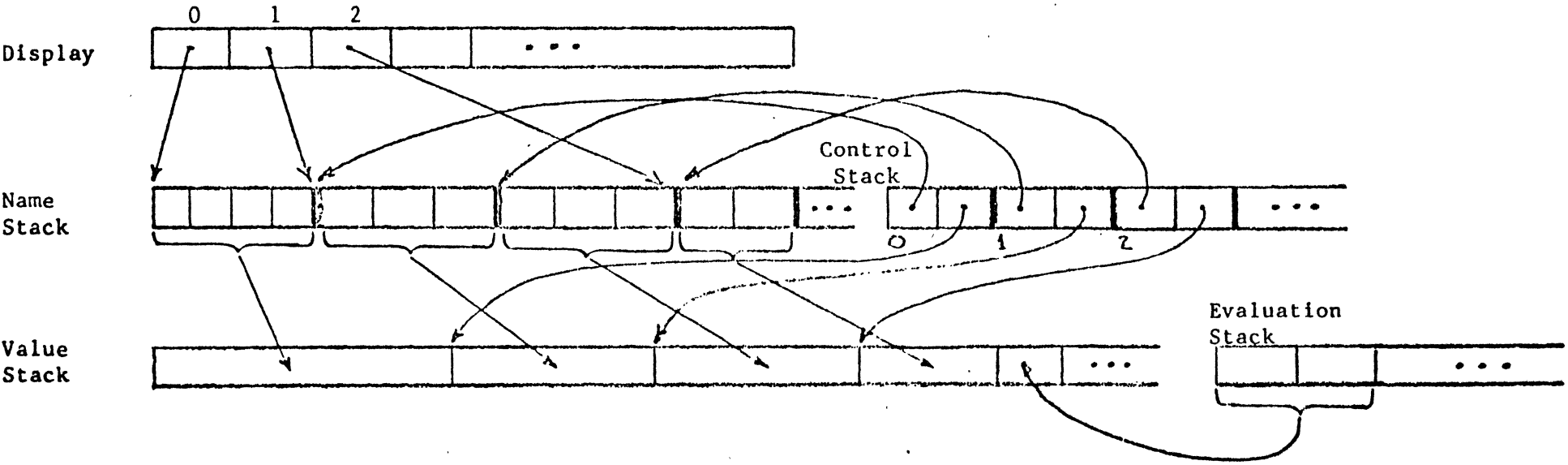

VALUE STACK
VSP

FIGURE 2a.

SDL STACK INTER-RELATIONSHIP



Control Stack Entry 1 describes a currently inactive lexic level.

FIGURE 2b.

SDL OPCODE STRUCTURE

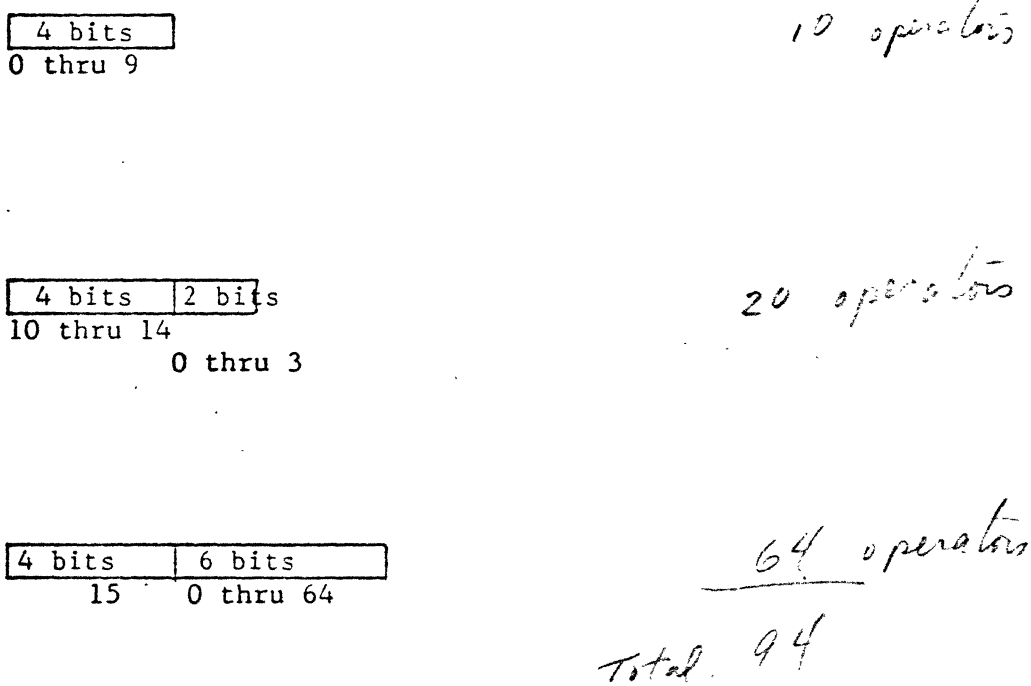


FIGURE 3.

MCP OPERATOR ENCODING

ENCODING METHOD	TOTAL BITS FOR MCP'S OPCODES	UTILIZATION IMPROVEMENT	DECODING PENALTY
HUFFMAN	172,346	43%	17.2%
SDL 4,6,10	184,966	39%	2.6%
8-BIT FIELD	301,248	0%	0.0%

FIGURE 4a.

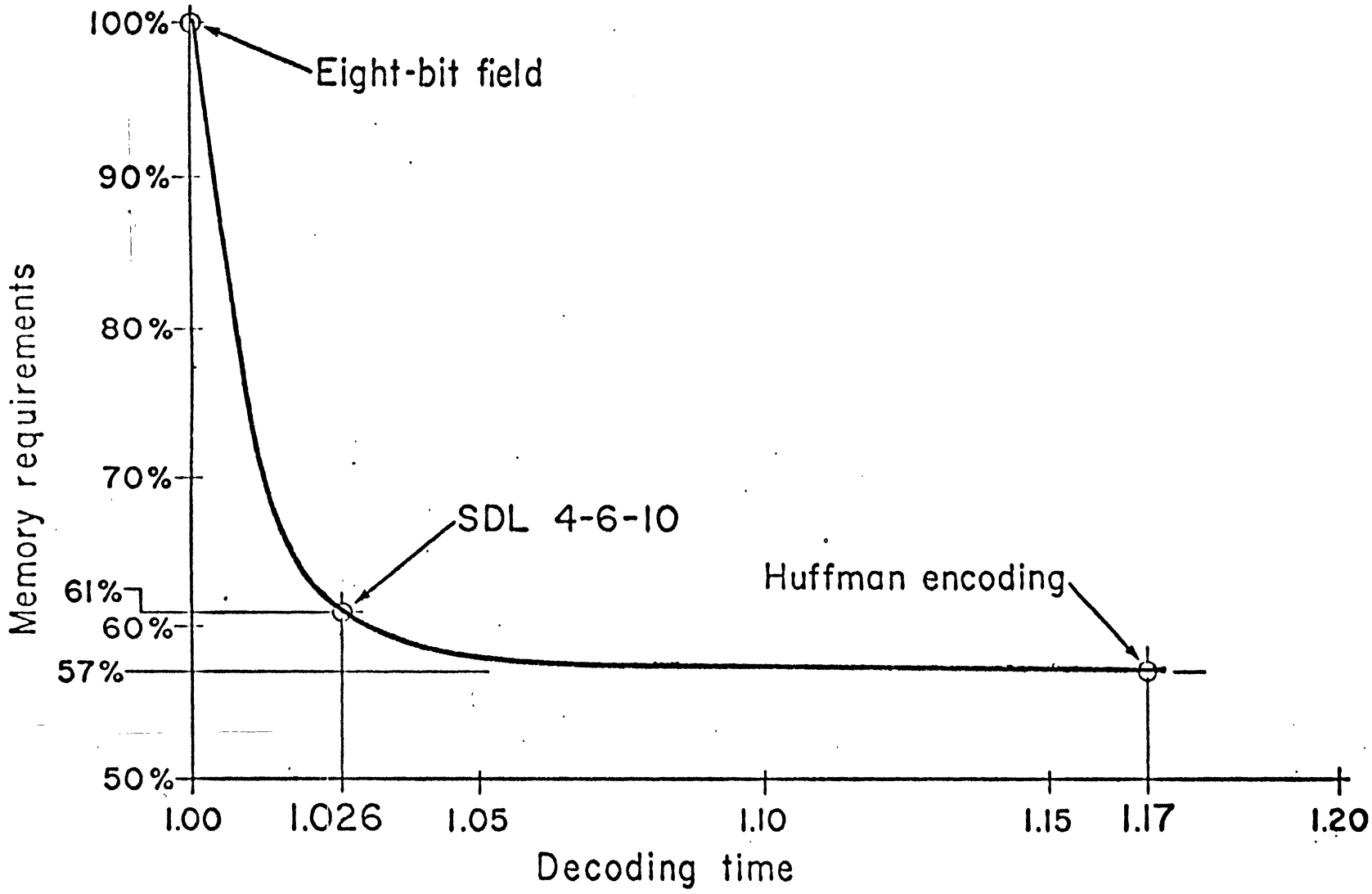
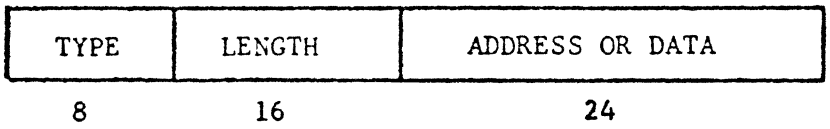

100%
90%
80%
70%
61%
60%
57%
50%
Memory requirements
Eight-bit field
SDL 4-6-10
Huffman encoding
1.00
1.026
1.05
1.10
1.15
1.17
1.20
Decoding time

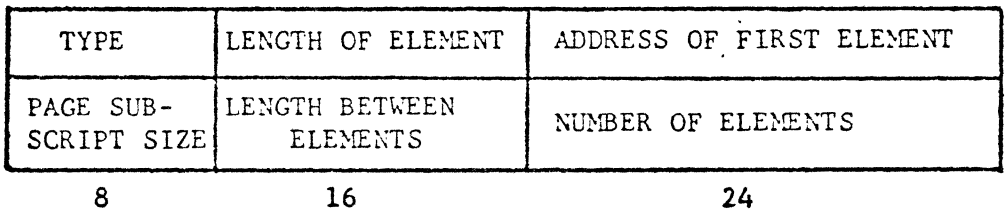
FIGURE 4b.

SDL DESCRIPTOR FORMATS

SIMPLE DESCRIPTOR:



ARRAY DESCRIPTOR:



TYPE FIELD:

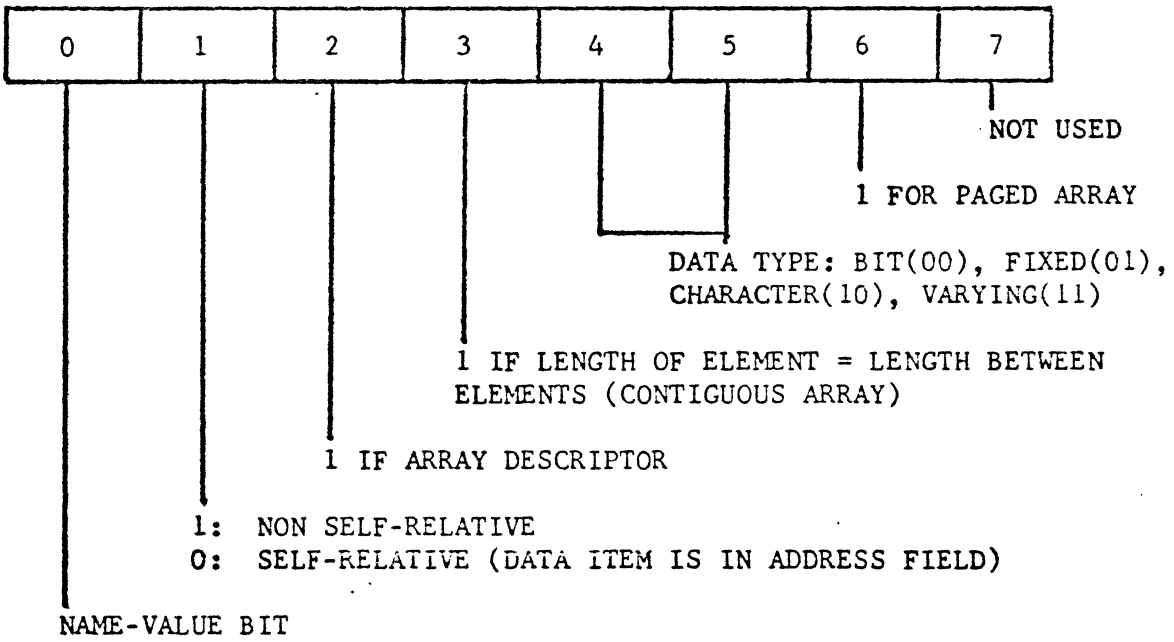


FIGURE 5.

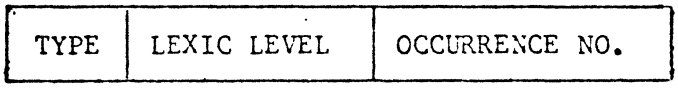
SDL CODE ADDRESS

TYPE	SEGMENT NO.	PAGE NO.	DISPLACEMENT
3	0 OR 6	0 OR 4	12,16,OR20

TYPE	SEGMENT NO.	PAGE NO.	DISPLACEMENT	TOTAL BITS
000	CURRENT	CURRENT	12 BITS	15
001	CURRENT	CURRENT	16 BITS	19
010	6 BITS	CURRENT	12 BITS	21
011	6 BITS	CURRENT	16 BITS	25
100	6 BITS	4 BITS	12 BITS	25
101	6 BITS	4 BITS	16 BITS	29
110	6 BITS	4 BITS	20 BITS	33
111	-	-	-	3

FIGURE 6.

SDL DATA ADDRESSES



TYPE	LEXIC LEVEL	OCCURRENCE NO.	TOTAL BITS
00	4 BITS	10 BITS	16
01	4 BITS	5 BITS	11
10	1 BIT *	10 BITS	13
11	1 BIT *	5 BITS	8

* 0: LEXIC LEVEL 0

1: CURRENT LEXIC LEVEL

FIGURE 7.

SDL CONSTRUCT DESCRIPTOR OPERATORS

IN-LINE DESCRIPTOR FORMAT:

TYPE	LENGTH	FILLER	LENGTH BETWEEN ELEMENTS	PAGE SUBSCRIPT SIZE	NUMBER OF ELEMENTS
8	6 OR 17	0,6,OR 17	0,6, OR 17	0 OR 8	0, 6, 17

6- OR 17-BIT FIELDS:

0	5 BITS

1	16 BITS

CONSTRUCT DESCRIPTOR OPERATORS:

OPERATOR	MNEMONIC	OPCODE	ARGUMENTS
BASE ZERO	CDBZ	1111 10 0100	D
LOCAL DATA	CDLD	1110 00	N,D1,...,DN
FROM PREVIOUS	CDPR	1110 10	N,D1,...,DN
FROM PREVIOUS AND ADD	CDAD	1110 01	N,D1,...,DN
FROM PREVIOUS AND MULTIPLY	CDMP	1111 10 0101	N,D1,...,DN
LEXIC LEVEL	CDLL	1111 10 0011	DA,D

WHERE D AND DI ARE IN-LINE DESCRIPTORS, AND DA IS A DATA ADDRESS (TYPE, LEXIC LEVEL, OCCURRENCE NUMBER)

FIGURE 8a.

SDL CONSTRUCT DESCRIPTOR ADDRESS CALCULATIONS

OPERATOR	ADDRESS
CDBZ	A' = 0
CDLD	A' = V
CDPR	A' = A + F
CDAD	A' = A + F + L
CDMP	A' = A + F + L + (E-1) × LB
CDLL	A' = ADDRESS(DA) + F

WHERE

A' IS THE NEW ADDRESS PART

V IS THE VALUE STACK POINTER

A IS THE ADDRESS PART OF THE PREVIOUS ENTRY IN THE NAME STACK

F IS THE IN-LINE FILLER VALUE, IF PRESENT

L IS THE LENGTH OF THE PREVIOUS ENTRY ON THE NAME STACK

E IS THE NUMBER-OF-ENTRIES PART OF THE PREVIOUS ENTRY ON THE NAME STACK

LB IS THE LENGTH-BETWEEN-ENTRIES PART OF THE PREVIOUS ENTRY ON THE NAME STACK

DA IS THE IN-LINE DATA ADDRESS

FIGURE 8b.

SDL CONTROL STATEMENT OPERATORS

OPERATOR	MNEMONIC	OPCODE	ARGUMENTS
CALL	CALL	0111	CA
IF-THEN	IFTH	1001	CA
IF-THEN-ELSE	IFEL	1101 10	AT,CA,CA
CASE	CASE	1111 01 0100	N,AT,CA1,...,CAN
UNDO	UNDO	1000	L
UNDO CONDITIONALLY	UNDC	1111 01 0011	L
CYCLE	CYCL	1110 11	D

WHERE

CA IS A CODE ADDRESS (TYPE, SEGMENT NUMBER, PAGE NUMBER, DISPLACEMENT)

AT IS THE CODE ADDRESS TYPE

N IS THE NUMBER OF CODE ADDRESSES

L IS THE NUMBER OF LEVELS TO UNDO

D IS THE NUMBER OF BITS OF DISPLACEMENT

FIGURE 9.

SDL PROCEDURE ENTRANCE AND EXIT OPERATORS

OPERATOR	MNEMONIC	OPCODE	ARGUMENTS
MARK STACK	MKS	1011 11	
MARK STACK AND UPDATE	MKU	1111 01 1111	LL
CALL	CALL	0111	CA
EXIT	EXIT	1101 11	L
UNDO	UNDO	1000	L
RETURN	RTRN	1111 01 0101	L,D

WHERE

LL IS THE ENTERED LEXIC LEVEL

CA IS A CODE ADDRESS

L IS THE NUMBER OF LEVELS TO REMOVE FROM THE PROGRAM POINTER STACK

D IS A TYPE, LENGTH PAIR

FIGURE 10.

SDL CONSTRUCT DESCRIPTOR FORMAL

OPERATOR	MNEMONIC	OPCODE	ARGUMENTS
CONSTRUCT DESCRIPTOR FORMAL	CDFM	1111 01 0001	L,E,D1,...,DN

WHERE

L IS THE ENTERED LEXIC LEVEL

E IS THE NUMBER OF 48-BIT ENTRIES ON THE EVALUATION STACK

DI ARE IN-LINE DESCRIPTOR TEMPLATES OF THE FORM:

TYPE	LENGTH	NUMBER OF ENTRIES
8	0,16	0,16

TYPE:

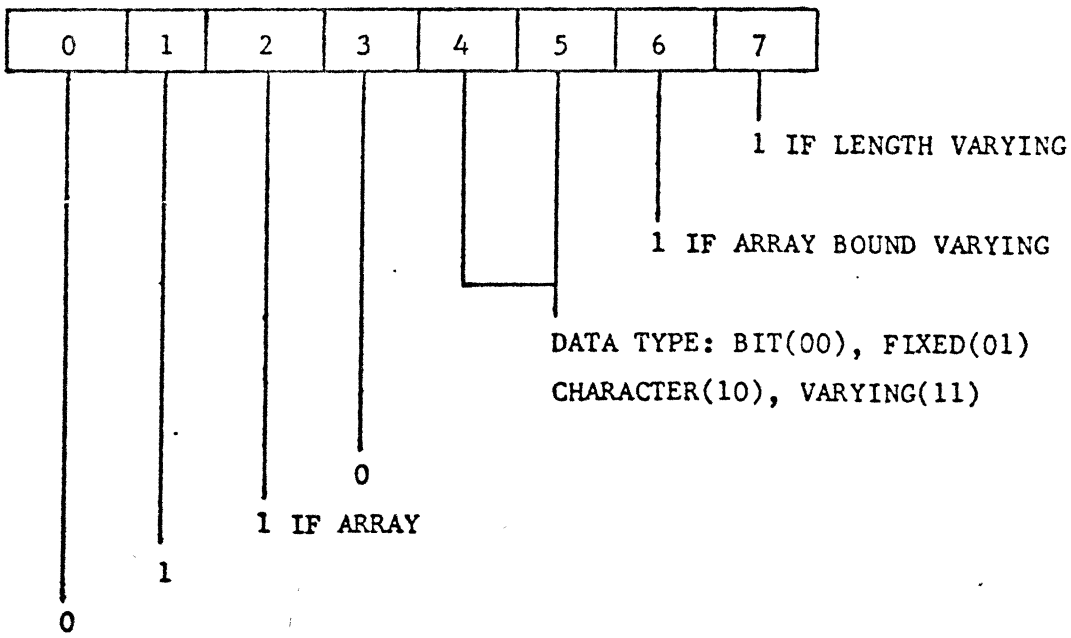


FIGURE 11.